



Universidade Nova de Lisboa

Faculdade de Ciências e Tecnologia

Departamento de Informática

Pattern Operators for Grid Environments

Maria Cecília Farias Lorga Gomes

Dissertação apresentada para a obtenção
do Grau de Doutor em Informática pela
Universidade Nova de Lisboa, Faculdade
de Ciências e Tecnologia.

Lisboa
(2007)

This dissertation was prepared under the supervision of
Professor José Cardoso e Cunha,
of the Faculdade de Ciências e Tecnologia,
Universidade Nova de Lisboa.

*To my parents,
and in memory of my uncle Dário*

Acknowledgements

I would like to express my gratitude to all those that, directly or indirectly, have contributed to make this thesis possible. In particular, I would like to thank my supervisor, Prof. Cardoso e Cunha for his invaluable technical and human support and for the confidence he had on me during the thesis time. Without him, this thesis could had never been possible. Moreover, I would like to express my gratitude to Dr. Omer Rana with whom I had the privilege to work with. Dr. Rana's remarkable professional expertise and human qualities enriched and enlarged both my technical and personal horizons. I would also like to thank Dr. Pedro Medeiros for his invaluable comments and help on this work, and for his true friendship and kindness.

Many thanks are also due to João Lourenço and Vítor Duarte for their help on many technical issues and their human support. I am honoured and lucky to have been sharing the same office with my friend Vítor who always gave me strength through many difficult times. Many thanks are also due to Hervé Paulino, Fernanda Barbosa, Rui Marques, Sérgio Duarte, Margarida Mamede, João Pires, and Jorge Cruz. Special gratefulness to my friends Anabela, Iara, Simone Ludwig, Ingo, and Michael, for all the support and useful advices.

The work over the Triana tool was only possible because of the invaluable help of Dr. Matthew Shields and Dr. Ian Wang, to whom I would like to express my gratitude. Special thanks are due to Ian Taylor for access to the Triana source code, and A. Nelson, N. White, P. Williams, and R. Philp, of the Galaxy Formation Group, Department of Physics and Astronomy, Cardiff University, UK, for the simulation data for the Galaxy application.

Finally, my very special thanks to my family. To my brothers and my sisters-in-law for their kindness, and to my nephews and nieces, Tiago, Ana Rita, Miguel, and Marta, for their endless love, energy, and joy for life. To my parents, in particular, whose wise words and kindness lightened my darkest days. They were always present whenever I needed them. I would also like to honor my two aunts Elvira Marta and Maria Amélia for their kind comfort.

I would also like to acknowledge the following institutions for their financial support:

Departamento de Informática and Faculdade de Ciências e Tecnologia of the Universidade Nova de Lisboa;
Centro de Informática e Tecnologias da Informação of the FCT/UNL;
Reitoria da Universidade Nova de Lisboa;
Distributed Collaborative Computing Group, Cardiff University, United Kingdom;
Fundação para a Ciência e Tecnologia;
Instituto de Cooperação Científica e Tecnológica Internacional;
French Embassy in Portugal;
European Union Commission through the Agentcities.NET and Coordina projects;
and the European Science Foundation, EURESCO.

Summary

The definition and programming of distributed applications has become a major research issue due to the increasing availability of (large scale) distributed platforms and the requirements posed by the economical globalization. However, such a task requires a huge effort due to the complexity of the distributed environments: large amount of users may communicate and share information across different authority domains; moreover, the “execution environment” or “computations” are dynamic since the number of users and the computational infrastructure change in time. Grid environments, in particular, promise to be an answer to deal with such complexity, by providing high performance execution support to large amount of users, and resource sharing across different organizations. Nevertheless, programming in Grid environments is still a difficult task. There is a lack of high level programming paradigms and support tools that may guide the application developer and allow reusability of state-of-the-art solutions.

Specifically, the main goal of the work presented in this thesis is to contribute to the simplification of the development cycle of applications for Grid environments by bringing structure and flexibility to three stages of that cycle through a common model. The stages are: the design phase, the execution phase, and the reconfiguration phase. The common model is based on the manipulation of patterns through pattern operators, and the division of both patterns and operators into two categories, namely structural and behavioural. Moreover, both structural and behavioural patterns are first class entities at each of the aforesaid stages. At the design phase, patterns can be manipulated like other first class entities such as components. This allows a more structured way to build applications by reusing and composing state-of-the-art patterns. At the execution phase, patterns are units of execution control: it is possible, for example, to start or stop and to resume the execution of a pattern as a single entity. At the reconfiguration phase, patterns can also be manipulated as single entities with the additional advantage that it is possible to perform a structural reconfiguration while keeping some of the behavioural constraints, and vice-versa. For example, it is possible to replace a behavioural pattern, which was applied to some structural pattern, with another behavioural pattern.

In this thesis, besides the proposal of the methodology for distributed application development, as sketched above, a definition of a relevant set of pattern operators was made. The methodology and the expressivity of the pattern operators were assessed through the development of several representative distributed applications. To support this validation, a prototype was designed and implemented, encompassing some relevant patterns and a significant part of the patterns operators defined. This prototype was based in the Triana environment; Triana supports the development and deployment of distributed applications in the Grid through a dataflow-based programming model. Additionally, this thesis also presents the analysis of a mapping of some operators for execution control onto the Distributed Resource Management Application API (DRMAA).

This assessment confirmed the suitability of the proposed model, as well as the generality and flexibility of the defined pattern operators.

Resumo

A concepção e a programação de aplicações distribuídas é cada vez mais um tema de intensa investigação, devido à crescente disponibilidade de plataformas distribuídas de grande escala e às solicitações resultantes da globalização económica e social. Contudo, o desenvolvimento das referidas aplicações requer um grande esforço por causa da complexidade inerente aos ambientes distribuídos: um grande número de utilizadores situados em diferentes domínios administrativos, podendo comunicar entre si e partilhar informação; além disso, o ambiente de execução das aplicações é dinâmico, uma vez que a plataforma computacional, o número de participantes, e a informação solicitada ou gerada, variam ao longo do tempo. Os ambientes de execução baseados em Grids¹ computacionais têm potencial para lidar com aquela complexidade, uma vez que disponibilizam uma plataforma de alto desempenho vocacionada para suportar múltiplos utilizadores e partilha de recursos em diferentes organizações. No entanto, programar no ambiente de uma Grid computacional é ainda uma tarefa difícil. Há falta de paradigmas de programação de alto nível que suportem a actividade do programador de aplicações, nomeadamente no aspecto da reutilização de componentes já existentes e testados, bem como das interacções entre os vários componentes que compõem uma aplicação.

Parte do ciclo de desenvolvimento de uma aplicação para uma Grid computacional é composto pelas fases de desenho, de execução e de reconfiguração. O principal objectivo desta dissertação é simplificar as actividades conduzidas neste ciclo através da proposta de um modelo de estruturação flexível e comum às três fases. Este modelo é baseado na manipulação de padrões (patterns) através da definição de operadores de padrões; os padrões e os operadores são divididos em duas categorias: estruturação e comportamento. Em particular, quer os padrões de estruturação quer os de comportamento são entidades de primeira ordem em cada uma das fases acima referidas. Na fase de desenho, os padrões podem ser manipulados como entidades de primeira ordem, tal como os componentes. Assim, uma forma mais estruturada de desenvolvimento de aplicações é suportada através da reutilização e da composição de padrões pré-existentes. Na fase de execução, os padrões são unidades de controlo da execução:

¹Designação derivada da analogia com a rede eléctrica "Power Grid".

é possível, por exemplo, lançar, parar e retomar a execução dos componentes computacionais que constituem um padrão como uma entidade única. Na fase de reconfiguração, os padrões podem também ser manipulados como entidades únicas, com a vantagem ser possível executar uma reconfiguração da estrutura, enquanto se assegura a manutenção das especificações de comportamento; o inverso é também possível, isto é, manter a estrutura e modificar o comportamento, por substituição do padrão de comportamento.

Nesta dissertação, além da proposta da metodologia de desenvolvimento de aplicações distribuídas acima esboçada, definiu-se um conjunto relevante de operadores de padrões. A metodologia e a definição dos operadores de padrões foram validadas através do desenvolvimento de um variado conjunto de aplicações distribuídas representativas. Para suportar esta validação, foi desenhado e implementado um protótipo de um ambiente de desenvolvimento de aplicações que suporta uma parte significativa do modelo desenvolvido. Este protótipo baseou-se no ambiente Triana, que suporta também o desenvolvimento de aplicações em Grids computacionais, assente num modelo de composição de componentes baseado em fluxos de dados. Foi também apresentada a análise de um mapeamento de alguns operadores de controlo da execução na especificação DRMAA (uma interface de programação de aplicações baseada nas funcionalidades de um gestor de recursos distribuídos).

Esta validação, bem como outros exemplos ilustrativos apresentados, permitiram confirmar a adequação do modelo proposto, bem como a aplicabilidade e a flexibilidade dos operadores de padrão definidos.

Contents

1	Introduction	1
1.1	Motivation	2
1.1.1	The Importance of Grid environments	2
1.1.2	Difficulties of Grid Application Development	3
1.1.3	Problem Solving Environments (PSEs)	6
1.2	The Need of High-Level Abstractions for Grid Application Development	11
1.2.1	Components and Services	11
1.2.2	Skeletons and Design Patterns	11
1.2.3	The Main Goal of this Work	12
1.3	The Proposed Model	13
1.3.1	Structural and Behavioural Patterns	13
1.3.2	Pattern Operators	17
1.3.3	A Methodology within the Model	20
1.3.4	Assisting Application Development in PSEs	21
1.4	Contributions of the Thesis	23
1.4.1	Work Approach	23
1.5	Dissertation Outline	23
2	Abstractions for Grid Programming	25
2.1	Introduction	26
2.2	General Solutions	27
2.2.1	Component Paradigm	27
2.2.2	Dynamic Reconfiguration and Adaptability	30
2.2.3	Coordination Paradigm	32
2.3	Solutions for Structure and Interaction Reusability	32
2.3.1	Skeletons	34
2.3.2	Patterns	36
2.4	Skeleton/Pattern-based Models and Systems	39
2.4.1	Skeleton-based Models and Systems	39
2.4.2	Pattern-based Models and Systems	45

2.5	Summary	47
3	Characteristics of the Model	51
3.1	Introduction	52
3.1.1	Structural and Behavioural Patterns	52
3.1.2	Structural and Behavioural Operators	54
3.1.3	The Basic Methodology	55
3.2	Pattern Templates	57
3.2.1	Structural Pattern Templates: Topological	58
3.2.2	Structural Pattern Templates: Non-Topological	60
3.2.3	Graphical Representation of Structural Pattern Templates	62
3.2.4	Behavioural Pattern Templates	63
3.2.5	Combining Behavioural and Structural Patterns	66
3.3	Operators	68
3.3.1	Operator Categories	69
3.3.2	Structuring and Grouping Operators	71
3.3.3	Inquiry Operators	74
3.3.4	Ownership Operators	76
3.3.5	Execution Operators	77
3.3.6	Global Coordination Operators	79
3.3.7	Pattern and Operator Summary	80
3.4	Summary	80
4	Pattern Operator Semantics	83
4.1	Introduction	84
4.2	Semantics of Structural Operators	84
4.2.1	Structuring Operators	85
4.2.2	Grouping Operators	97
4.3	Sequences of Structural Operators	106
4.3.1	Sequences Including the Replicate, Replace, or Reshape Operators	106
4.3.2	Sequential Application of Extend, Increase/Decrease, and Reduce	108
4.3.3	Structural Operation of Hierarchical Pattern Templates	112
4.4	Semantics of Behavioural Operators	115
4.4.1	The CO_OPN/2 Formalism	116
4.4.2	Start and Terminate Operators	120
4.4.3	Stop and Resume Operators	121
4.4.4	Repeat and TerminateRepeat Operators	122
4.4.5	Limit Operator	124
4.4.6	Restart and TerminateRestart Operators	125
4.4.7	Log Related Operators	126
4.5	Sequences of Behavioural Operators	129

4.5.1	Common Sequences and Compound Operators	129
4.5.2	Controlling Individual Executions in the Context of the Restart/Repeat Operators	130
4.6	Summary	133
5	Towards Pattern-based Reconfiguration	135
5.1	Introduction	136
5.2	The Methodology Revisited	136
5.2.1	Methodology Steps	137
5.2.2	Operating a Pattern Template (SB-PT)	139
5.2.3	Operating Component Instantiated Structural Patterns (CISPs) .	150
5.2.4	Operating Pattern Instances (PIs)	155
5.3	Reconfiguration	163
5.3.1	Reconfiguration Options	164
5.3.2	Reconfiguration Examples	166
5.4	Summary	168
6	The Architecture and its Implementation	169
6.1	Introduction	170
6.2	The Architecture Supporting the Model	170
6.2.1	Application Configuration and Execution Control	172
6.2.2	Application Reconfiguration	174
6.3	An Instance of the Architecture: Implementation over Triana	177
6.3.1	The Specific Architecture	177
6.3.2	The Triana Environment	181
6.4	Patterns and Operators in Triana	185
6.4.1	Structural Patterns and Operators in the Triana GUI	187
6.4.2	Scripts of Structural Patterns and Operators	190
6.4.3	Execution Control from the Triana GUI and from Scripts	196
6.4.4	Implementation in Triana	202
6.5	Mapping to the DRMAA API	208
6.5.1	Start and Terminate Behavioural Operators	210
6.5.2	Stop and Resume Behavioural Operators	212
6.5.3	Restart and Repeat Behavioural Operators	213
6.5.4	Limit Behavioural Pattern	215
6.6	Summary	216
7	Validation	219
7.1	Introduction	220
7.1.1	Conceptual Examples	220
7.1.2	Examples in Triana	220

7.2	Configuring Distributed Systems Topologies	221
7.2.1	Basic Topologies	221
7.2.2	Hybrid Topologies	224
7.3	Configuring a Problem Solving Environment	226
7.3.1	A Typical PSE Example	226
7.3.2	Structural Patterns in Use	228
7.3.3	Behavioural Patterns in Use	229
7.3.4	Structural Operations	231
7.3.5	Behavioural Operators in Use	234
7.3.6	Reconfiguration Scenarios	236
7.4	Skeleton Modelling	244
7.4.1	Mapping P3L Skeletons to Structural and Behavioural Patterns .	244
7.4.2	Modelling a P3L Example	249
7.4.3	Reconfiguring the P3L Example	253
7.5	Analysis of Gravitational Waves	259
7.5.1	Simulation in Triana	259
7.5.2	Configuration and Execution through a Script	262
7.5.3	Simulating Regular Production of Data	264
7.6	Galaxy Formation Example	269
7.6.1	Alternative Configuration	271
7.6.2	Introducing Execution Control and Reconfiguration	272
7.7	Simulating Flexible Information Retrieval and Processing	281
7.7.1	Database Access	281
7.7.2	First Structural Reconfiguration: Accessing a New Tool	283
7.7.3	Second Structural Reconfiguration: Pattern Replacement	286
7.8	Summary	288
8	Conclusions and Future Work	289
8.1	Conclusions	290
8.1.1	Contributions of the Thesis	290
8.2	Future Work	293

List of Figures

1.1	<i>A typical logical architecture for a distributed/parallel PSE consisting of three components – a distributed simulator, a visualisation and a control component.</i>	8
1.2	<i>A generic PSE.</i>	15
1.3	<i>Example of the identification of Structural and Behavioural Patterns in a PSE.</i>	15
1.4	<i>Pattern-based configuration of the example in Figure 1.3. All elements of the Structural Patterns are already instantiated to the necessary tools/services.</i>	18
1.5	<i>Launching periodically the execution of the PSE.</i>	20
1.6	<i>New configuration of the PSE (e.g. providing support for an additional user of the Steering Interface, i.e. “proxy3”).</i>	20
1.7	<i>The software life-cycle of application development in PSEs (left column) and the mapping of the application of the pattern and operators model to that life-cycle (right column).</i>	22
2.1	<i>Useful paradigms and techniques and their characteristics.</i>	27
3.1	<i>Relating the used pattern definitions.</i>	54
3.2	<i>The basic steps of the methodology.</i>	56
3.3	<i>The Star pattern.</i>	58
3.4	<i>The Pipeline pattern.</i>	59
3.5	<i>The Connector pattern.</i>	59
3.6	<i>The Ring pattern.</i>	60
3.7	<i>The Adapter pattern [9].</i>	61
3.8	<i>The Facade design pattern [9]. Example: the “Facade” provides a unified interface for accessing domains in the Grid environment, redirecting the calls to services like “discover” and “execute”.</i>	61
3.9	<i>The Proxy pattern [9].</i>	62
3.10	<i>Graphical representation of examples of Structural Pattern Templates (S-PTs).</i>	62
3.11	<i>A Sequence diagram for the Mobile Agent/Itinerary pattern defining a possible itinerary for the component.</i>	66
3.12	<i>A Remote Evaluation B-PT applied to a Facade PT.</i>	67

3.13	<i>Two Behavioural Pattern Templates, namely Client/Server and Observer B-PTs, applied to a Star PT.</i>	67
4.1	<i>The creation of three S-PTs, namely a Star ("starPT"), a Facade ("facadePT"), and an Adapter ("adapterPT")</i>	86
4.2	<i>Examples of the Reshape operator over a Pipeline and a Proxy Pattern Templates.</i>	87
4.3	<i>Examples of the Increase operator applied to Pipeline, Proxy, and Facade Pattern S-PTs.</i>	89
4.4	<i>Examples of the Decrease operator applied to Pipeline, Proxy, and Facade Pattern S-PTs.</i>	90
4.5	<i>Examples of the second application versions of the Increase and Decrease operators upon a Pipeline S-PT.</i>	90
4.6	<i>Examples of the "Extend(P)" operator over cases of the Proxy, Adapter, and Facade Pattern Templates.</i>	91
4.7	<i>Example of the "Extend(element, P)" operator over one Facade Template.</i>	92
4.8	<i>Example of the "Extend(element, P)" operator over one case of Adapter Template.</i>	93
4.9	<i>Different ways of applying the "Extend(element, P)" operator to a Proxy Template.</i>	94
4.10	<i>Examples of the application of both versions of the Reduce operator to one Proxy Template.</i>	95
4.11	<i>Examples of both versions of the Reduce operator over one Facade Template.</i>	96
4.12	<i>Examples of both versions of the Reduce operator applied to an Adapter Template.</i>	97
4.13	<i>The pattern templates "ringPT" and "startPT" are grouped through the Group operator, and the resulting aggregate is named "groupPT".</i>	98
4.14	<i>The group "groupPT" is dissolved through the Ungroup operator.</i>	99
4.15	<i>Adding a extra pattern template to the aggregate "groupPT".</i>	99
4.16	<i>Merging of groups "group1PT" and "group2PT" through the Group operator, producing the aggregate "group1PT".</i>	100
4.17	<i>An example of a pipeline template with an embedded pattern (a star) in the leftmost stage ("cph1"). This hierarchic pattern template is built through the Embed operator.</i>	100
4.18	<i>Examples of possible connections between the embedded pattern and the enclosing pattern.</i>	101
4.19	<i>Extracting pattern "starPT" from within the first stage of the pattern "pipelinePT".</i>	102
4.20	<i>Embedding an adapter template into a proxy template in the position of the "real subject".</i>	102
4.21	<i>Embedding a group ("group1PT") into another ("group2PT").</i>	103

4.22	<i>Embedding the same pattern template into two Hierarchical Pattern Templates. In both examples, the "adapterPT" is embedded in the "nucleus" of a "starPT", but in one case (upper part of the Figure) this latter pattern is included in a group, whereas in the other, the "starPT" is embedded in the "realsubject" position of the "proxyPT".</i>	104
4.23	<i>Extracting a pattern template from a Hierarchical Pattern Template, namely a starPT is removed from within the "realsubject" of a proxyPT, and the "real-subject" gets uninstantiated.</i>	105
4.24	<i>Extracting a pattern template from a Hierarchical Pattern Template, specifically, the adapterPT is extracted from the "nucleus" of the starPT which is located in the position of the "real subject" in the proxyPT. Consequently, the "nucleus" of the starPT gets uninstantiated.</i>	105
4.25	<i>Building a new similar PT to the ProxyPT but where a ringPT is embedded in the "nucleus" of the starPT (instead of the adapterPT).</i>	107
4.26	<i>Transforming an embedded pattern into another through the Reshape operator.</i>	108
4.27	<i>Applying the Extend and Increase operators in sequence to a Proxy PT.</i>	108
4.28	<i>Applying the Extend and Increase operators in sequence to a Facade PT.</i>	110
4.29	<i>Applying the two versions of the Reduce operator to the "proxyPT" template.</i>	111
4.30	<i>Applying the two versions of the Reduce operator to the "facadePT" template.</i>	111
4.31	<i>Applying the Increase and Decrease operators to a pipeline hierarchic pattern that contains an embedded pattern in the first stage.</i>	113
4.32	<i>Applying the Extend operator to a proxy hierarchic pattern that contains an embedded pattern in the "realsubject" element.</i>	114
4.33	<i>Applying the Reduce operators to a facade hierarchic pattern that contains an embedded pattern in the outmost facade element (i.e. "facade2").</i>	115
4.34	<i>An example of a CO_OPN/2 object with its behaviour modelled through a Petri-Net.</i>	116
4.35	<i>An example of a synchronised call between two CO_OPN/2 objects.</i>	117
4.36	<i>An example of a CO_OPN/2 context with two objects.</i>	118
4.37	<i>Example of the Start and Terminate operators over a pipeline pattern instance.</i>	120
4.38	<i>Example of the Stop and Resume operators applied to a pipeline instance.</i>	121
4.39	<i>Example of the Repeat and TerminateRepeat operators applied to a pipeline instance.</i>	123
4.40	<i>Example of the Limit operator applied to a pipeline instance.</i>	124
4.41	<i>Example of the Restart and TerminateRestart operators applied to a pipeline instance.</i>	125
4.42	<i>Example of the Log, TerminateLog, SeqLog, TermSeqLog, and ResumeLog operators applied to a pipeline instance.</i>	127
5.1	<i>Relating the used pattern definitions.</i>	137
5.2	<i>Methodology steps for application configuration and execution control.</i>	138

5.3	<i>Applying a single Behavioural Pattern to all elements of a Structural Pattern forming a Regular SB-PT.</i>	140
5.4	<i>Defining the behavioural role of one specific element within a pattern (and the adding of other necessary behavioural annotations). Since that behavioural role pertains a different Behavioural Pattern than the one already applied to the pattern, the result is an Heterogeneous SB-PT.</i>	140
5.5	<i>Replicating a SB-PT in two ways: a) considering it as a first class entity ("facadeSB-PT"); b) acting only over the Structural Pattern Template included in the SB-PT ("facadeS-PT").</i>	144
5.6	<i>Augmenting the number of component-place holders of a SB-PT in two ways: a) considering it as a first class entity ("facadeSB-PT"); b) acting only over the Structural Pattern Template included in the SB-PT ("facadeS-PT").</i>	146
5.7	<i>Extending two SB-PT in two ways: a) considering it as a first class entity ("proxySB-PT"); b) acting only over the Structural Pattern Template included in the SB-PT ("facadeS-PT").</i>	147
5.8	<i>Applying the Reduce operator to a SB-PT.</i>	148
5.9	<i>Modifying the behavioural annotations of a SB-PT considered as a first class entity ("facadeSB-PT").</i>	149
5.10	<i>Instantiation of the component place-holder "cph1" of the pattern "facade-CISP" to the "Resource management" component.</i>	150
5.11	<i>Extending a Component Instantiated Structural Pattern, namely "facadeCISP"</i>	151
5.12	<i>Increasing a Component Instantiated Structural Pattern ("facadeCISP") by two component place-holders.</i>	152
5.13	<i>Increasing a component instantiated Pipeline ("pipelineCISP) by two component place-holders inserted after element "FFT".</i>	152
5.14	<i>Application of the Decrease operator. The first example (upper part of the Figure) presents a case of reducing the number of component place-holders from a Partial CISP, namely, a partially instantiated Facade. The second example shows the usage of the Decrease operator to eliminate a set of elements from a Partial CISP ("pipelineCISP") starting at a specific element, namely the "Gaussian" element.</i>	154
5.15	<i>Elimination of one particular element of the "facadeCISP", namely "Scientific-tool".</i>	154
5.16	<i>Increasing a Regular FC-PI by one element, namely a pipeline pattern combined with the Streaming Behavioural Pattern.</i>	157
5.17	<i>Increasing an Heterogeneous PI by one element, namely a star pattern combined with the Observer and Client/Server Behavioural Patterns.</i>	158

5.18	<i>Decreasing a partially instantiated Heterogeneous PI by two component place-holders. Although it was requested the deletion of three CPHs, only the two existing CPHs are deleted. All behavioural annotations pertaining to those components are also eliminated.</i>	159
5.19	<i>Decreasing two PIs by one element. On top, the element “FFT” is removed from the Regular PI “pipelinePI”. On bottom, the element “Newscenter” is removed from the Heterogeneous PI “starPI”.</i>	160
5.20	<i>Applying the Extend operator to a PI (“adaptLegacyPI”). At the top, the structure is augmented disregarding the applied Behavioural Pattern. At the bottom, the PI is operated as a Regular FCSB-PT which results in the automatic annotation of the new element with a role within the applied Behavioural Pattern.</i>	161
5.21	<i>Applying the Reduce operator to a PI.</i>	162
5.22	<i>Changing the behavioural annotations of two Regular PIs.</i>	163
5.23	<i>Summary of the possible steps for reconfiguring a running application.</i>	164
5.24	<i>The dynamic reconfiguration of a Regular Pattern Instance representing a Grid service.</i>	167
5.25	<i>Building a dynamic itinerary for an Agent</i>	167
5.26	<i>Reconfiguring a Pattern Instance whose execution needs to be stopped.</i>	168
6.1	<i>A generic architecture that supports the model based on Patterns and Operators.</i>	171
6.2	<i>The necessary steps to configure and execute an application using patterns and pattern operators. Please read the Figure starting from the bottom.</i>	173
6.3	<i>Application of the Increase Structural Operator at development time and after step 4 in Figure 6.2, in order to instantiate application App5 as the last stage of the pipeline.</i>	175
6.4	<i>Modifying the control dependencies within the pattern, after the application in Figure 6.3 is executing.</i>	176
6.5	<i>The specific architecture, based on the Triana environment, which supports the patterns/operators model. The shaded elements in the upper layer are the result of the work presented in this dissertation.</i>	178
6.6	<i>The Triana’s Graphical User Interface.</i>	181
6.7	<i>A simplified vision of Triana’s distribution model.</i>	184
6.8	<i>The inclusion of Patterns and Operators into the Triana Environment.</i>	186
6.9	<i>The Triana’s Graphical User Interface.</i>	187
6.10	<i>Initialisation of Pattern Template.</i>	187
6.11	<i>Application of the Embed Structural Pattern to the Ring Pattern Template.</i>	188
6.12	<i>Instantiation of the DummyUnit component place-holder to the AccumStat Unit.</i>	189
6.13	<i>A Ring pattern fully instantiated at the outmost level.</i>	190
6.14	<i>A script with structural operations is associated to a particular pattern.</i>	190

6.15	<i>General structural manipulation from a script defined in EBNF. The presented actions (nonterminal EBNF elements) may be interleaved and applied as many times as necessary. A terminal element defining the end of script processing is omitted for simplification reasons.</i>	191
6.16	<i>Structural Operators.</i>	192
6.17	<i>EBNF definition of the usage of the Embed Structural Operator from a script.</i>	192
6.18	<i>EBNF definition of the names of the component place-holders within a Pattern Template.</i>	192
6.19	<i>EBNF graph for the “Instantiate_DummyUnit” nonterminal element in the graph in Figure 6.15.</i>	193
6.20	<i>EBNF graph for the “Create_and_Embed_Other_Patterns” nonterminal element in the graph in Figure 6.15.</i>	194
6.21	<i>EBNF graph for the “Create_Pattern” nonterminal element in the graph in Figure 6.20.</i>	194
6.22	<i>EBNF graph for the “Run _Structural _Script” nonterminal element in the EBNF graph in Figure 6.20.</i>	195
6.23	<i>EBNF graph for the “SetApplication _Parameter” nonterminal element in the EBNF graph in Figure 6.22.</i>	195
6.24	<i>The data and control flow connections in a (Hierarchical) Pattern Instance.</i>	198
6.25	<i>The parameter panel representing the Execution Operators and their arguments. The Restart Operator is selected to launch the periodic execution every 10000 milliseconds.</i>	198
6.26	<i>Application of the Terminate operator.</i>	199
6.27	<i>Execution debug information generated upon application of the Terminate operator to a pattern-based application ruled by the Restart operator.</i>	199
6.28	<i>EBNF graph for the Execution Operators.</i>	200
6.29	<i>EBNF graph for the execution control of pattern-based applications.</i>	200
6.30	<i>EBNF graph for explicit execution control including the usage of trigger nodes.</i>	200
6.31	<i>Activating a trigger node from a Pattern Instance’s parameter panel.</i>	201
6.32	<i>Definition of a Pattern Instance.</i>	203
6.33	<i>UML simplified description of some Triana classes.</i>	204
6.34	<i>Simplified UML definition of the classes for creating and manipulating Pattern Templates and Instances through Structural and Behavioural Operators, respectively. The definition includes a particular example of the Ring pattern template.</i>	205
6.35	<i>The data and control flow connections in a (Hierarchical) Pattern Instance.</i>	207
6.36	<i>A Streaming (data-flow) Behavioural Pattern combined with a Pipeline Structural Pattern. Figure A shows the entities before the execution of the Pattern Instance. Figure B shows the jobs created by a DRMS to support the execution of the applications (“App1”.. “App3”) in the “Pipeline” Pattern Instance.</i>	208

6.37	<i>DRMAA mapping of the Start operator.</i>	210
6.38	<i>DRMAA mapping of the Terminate operator.</i>	212
6.39	<i>DRMAA mapping of the Stop operator.</i>	212
6.40	<i>DRMAA mapping of the Resume operator.</i>	213
6.41	<i>DRMAA mapping of the Repeat operator.</i>	214
6.42	<i>DRMAA mapping of the Limit operator.</i>	215
7.1	<i>The Centralised and Ring distributed systems topologies.</i>	221
7.2	<i>The Hierarchical distributed systems topology (c) and its modelling through the Star Structural Pattern Template manipulated by Structural Operators.</i>	223
7.3	<i>The Decentralised topology.</i>	224
7.4	<i>The hybrid Centralised+Ring topology, and its configuration by embedding a Ring Pattern Template into the nucleus of a Star Pattern Template.</i>	224
7.5	<i>The hybrid Centralised+Centralised topology, and its configuration by the combination of the Star Structural Pattern and the Client/Server Behavioural Pattern.</i>	225
7.6	<i>The hybrid Centralised+Decentralised topology.</i>	226
7.7	<i>A PSE supporting the active steering of a Problem Solver. The arrows represent the flow of data.</i>	226
7.8	<i>The Monitoring service is stopped and consequently the Steering interface also stops. The output data is not lost because it is being saved in the Database system.</i>	227
7.9	<i>The initial Monitoring service is replaced with a more complex one (Monitoring and Statistics service), which is activated to continue the filtering of the output data.</i>	227
7.10	<i>After the Problem Solver terminates its execution, data can be re-analysed.</i>	228
7.11	<i>Identification of the Ring and Pipeline patterns in the PSE example.</i>	228
7.12	<i>Identification of the Star, Adapter, and Proxy patterns in the PSE example.</i>	229
7.13	<i>Combination of the Ring pattern with the Producer/Consumer Behavioural Pattern, and the Pipeline pattern with the Streaming Behavioural Pattern.</i>	230
7.14	<i>Combination of: the Star SP with the Master/Slave Behavioural Pattern; the Adapter SP and the Service Adapter Behavioural Pattern; and the Proxy SP with the Client/Server Behavioural Pattern.</i>	231
7.15	<i>Initial steps for building the PSE depicted in Figure 7.7.</i>	232
7.16	<i>Final steps for building the PSE depicted in Figure 7.7.</i>	233
7.17	<i>The final configuration for the PSE example.</i>	234
7.18	<i>Applying the Terminate Behavioural Operator to replace the embedded “MonitoringSo” Pattern Instance (PI) for the “Mon itStatSo” pattern.</i>	237
7.19	<i>Applying the Start Behavioural Operator to the “Mon itStatSo” pattern to continue the execution.</i>	238
7.20	<i>Execution suspension of the embedded “SteeringInt” Pattern Instance through the Stop Behavioural Operator and replacement of its Behavioural Pattern.</i>	239

7.21	<i>Resuming the execution of the embedded “SteeringInt” PI with the definition that the user “Pattern Controller” may manipulate this pattern with the Increase/Decrease and Instantiate operators.</i>	239
7.22	<i>Incrementing the number of proxies in the embedded “SteeringInt” providing access to the “Steering Interface” to an extra (passive) user.</i>	240
7.23	<i>New configuration for analysis of the PSE generated data formerly saved at the Database system.</i>	242
7.24	<i>A new component is directly connected to the Problem Solver. This configuration is based on the one presented in Figure 7.17.</i>	243
7.25	<i>Two Control Parallel skeletons: a) sequential and c) loop; and one Task Parallel skeleton: b) pipeline.</i>	245
7.26	<i>Modelling the farm Task Parallel skeleton.</i>	246
7.27	<i>The “map” and “comp” Data Parallel skeletons.</i>	247
7.28	<i>Modelling the reduce Data Parallel skeleton.</i>	248
7.29	<i>A P3L example [151] composed of four DP tasks interacting according to the composition of a farm TP and a pipeline TP.</i>	250
7.30	<i>One case of modelling the P3L example in Figure 7.29 using the Pattern/Operator model.</i>	250
7.31	<i>One possible reconfiguration of the pattern-based example in Figure 7.30 . . .</i>	254
7.32	<i>Two reconfiguration scenarios for the modelling presented in Figure 7.31. . . .</i>	256
7.33	<i>A P3L pipeline with two iterated stages. Results of stage 3 are fed back to stage 2 [151].</i>	257
7.34	<i>Modelling the P3L example in Figure 7.33 using the Pattern/Operator model. .</i>	257
7.35	<i>A simple example in the area of gravitational wave experiments.</i>	259
7.36	<i>Initialisation of a Star PT.</i>	260
7.37	<i>Addition of one satellite to a Star PT.</i>	260
7.38	<i>A Star PT with three satellites and a Pipeline PT with three elements.</i>	261
7.39	<i>Application of the Embed Structural Operator to the Star PT.</i>	261
7.40	<i>Instantiation of a Unit.</i>	262
7.41	<i>Final configuration.</i>	262
7.42	<i>Execution results.</i>	263
7.43	<i>Application of a script to a pattern template.</i>	264
7.44	<i>Debug window showing the application of the Restart Behavioural operator. . .</i>	265
7.45	<i>A simple simulation of regular production of data by the gravitational wave detection service.</i>	266
7.46	<i>Producing different waves every 10 seconds.</i>	266
7.47	<i>Two different waves produced at two consecutive execution steps.</i>	267
7.48	<i>Selection of the Terminate Behavioural operator.</i>	267
7.49	<i>The debug window showing the execution of the Terminate Behavioural operator.</i>	267

7.50	<i>Applying the Repeat Behavioural Pattern for launching the execution ten consecutive times.</i>	268
7.51	<i>The animation is supported by a pipeline PT which is embedded in the nucleus of the star PT.</i>	269
7.52	<i>An example of a component place holder instantiation.</i>	270
7.53	<i>A possible final configuration for the image processing of the “Galaxy Formation example”.</i>	271
7.54	<i>Execution snapshot for the selected configuration.</i>	272
7.55	<i>Parallel animation execution with different view points.</i>	272
7.56	<i>Detail of the stage named Pipeline in the Ring PT from Figure 7.55</i>	273
7.57	<i>Configuration and execution of the Galaxy simulation example through the script named “GalaxyExecCtrl”.</i>	274
7.58	<i>The ImgProjection pipeline.</i>	275
7.59	<i>The ImgProcessing pipeline.</i>	276
7.60	<i>The ImgAnalysis pipeline.</i>	276
7.61	<i>The Star Structural Pattern supporting the configuration of the Galaxy example.</i>	276
7.62	<i>Triggering the EnhContrast unit to consume the next 2D image.</i>	277
7.63	<i>Triggering the EnhContrast and CountBlobs units.</i>	278
7.64	<i>An execution step of the Galaxy example.</i>	279
7.65	<i>The results at both satellites in the next execution step after the one presented in Figure 7.64.</i>	280
7.66	<i>The configuration of the ImgProjection pipeline for a step-by-step execution. . .</i>	280
7.67	<i>The Facade Structural Pattern Template.</i>	282
7.68	<i>Configuration supporting the request of information to two sub-systems. . . .</i>	282
7.69	<i>Two Pipeline Structural Patterns supporting the configuration of two sub-systems for database access and output data analysis/processing.</i>	283
7.70	<i>The parameter panel of DBExplore, a database inquire tool available in Triana. .</i>	284
7.71	<i>Application of the Extend Structural Operator to the Facade Structural Pattern.</i>	284
7.72	<i>Result configuration of the action in Figure 7.71.</i>	285
7.73	<i>The innermost Facade acting as a subsystem of Facade1 in Figure 7.72. . . .</i>	286
7.74	<i>An Adapter Structural pattern providing access to a simulation tool. The Adapter pattern will replace Facade1.</i>	286
7.75	<i>The configuration after the application of the Replace Structural Operator described in Figure 7.74.</i>	287
7.76	<i>Another possible configuration where the client application, the Requester, receives processed data it has requested.</i>	287

List of Tables

1.1	<i>Issues addressed by PSEs.</i>	7
3.1	<i>Pattern Templates and Operator Summary.</i>	81
4.1	<i>Applicability of Structural Operators to Topological and Non-topological Structural Pattern Templates</i>	84

1

Introduction

Contents

1.1	Motivation	2
1.2	The Need of High-Level Abstractions for Grid Application Development	11
1.3	The Proposed Model	13
1.4	Contributions of the Thesis	23
1.5	Dissertation Outline	23

This chapter presents the motivation for providing high level software abstractions to aid in the construction of parallel and distributed applications, namely for Grid environments; enumerates the main contributions of this thesis; and presents an outline of the dissertation, with a brief summary of each of the following chapters.

1.1 Motivation

In this section, the motivation for this work is presented, namely, we highlight the need of adequate programming abstractions to support the development of distributed applications. Firstly, the main difficulties related to Grid programming are considered, followed by a discussion on the need of programming/software abstractions and environments for Grid computing.

The goal of this work is subsequently presented, namely to contribute to a development environment based on high-level abstractions, with a specific focus on support for structured and flexible composition for application construction, reconfiguration, and execution control. To that extent, this chapter introduces the main characteristics of the proposed approach underlying the above goal, and which are further discussed throughout this thesis.

1.1.1 The Importance of Grid environments

Large-scale distributed applications have increased in importance in the last years as a result of the Internet expansion and economical globalisation. People communicate globally more than ever, discovering the benefits of distributed environments and, at the same time, continuously demanding better capabilities provided by those environments. Therefore, there is a strong need for applications that reliably support large amounts of users independently from their location, support collaboration, span organisations' boundaries, and provide the user with adequate qualities of service. In this context, the growing importance of Grid environments results from their current and planned features for promising development for enabling such large-scale distributed applications.

Grids are highly heterogeneous, complex, and dynamic distributed systems providing large number of users the additional possibility of high computational power and access to large amounts of data [50,96]. Such facilities were not easily accessible or combinable before the Grid era. Nowadays, and through Grid environments, the user has, on one hand, access to a high number of very different resources such as high-performance computing and networks, storage systems, intelligent sensors, and many specific scientific instruments and systems; on the other hand, many of those resources are already accessed through standard services which aim to provide a common basis for application deployment.

As dynamic environments, Grids' operating characteristics can change significantly over the lifetime of a single application, for example, with resources being added and removed. Given the large number of organisations that can benefit from and contribute to the Grid, Grid platforms span different administrative domains in the worldwide context. Namely, Grid environments have to support the dynamic formation of virtual organisations, and also their modification in order to support different goals at

different times.

Grid Applications

Over the years, the motivation to exploit Grid environments has been increasing, both in science and business domains. Nowadays, Grids are increasingly targeted at non-academic areas such as business applications in the domains of Life Sciences, Electronic Design, Financial services, and Aerospace and Film industries [197–199].

Initially, the Grid concept was mainly motivated by developments in the area of High-Performance Computing and was aimed to support computational scientists on their efforts to enable larger engineering and scientific applications. Examples, among others, are projects such as the DataGrid [57] and MyGrid [56] that provide scientific platforms that simplify the application development in specialised domains such as High-Energy Physics, Astrophysics, Biology, Earth observation, etc.

Present and future Grid applications in science and engineering aspire to hide the complexity of the underlying execution platforms and integrating both specific and general purpose tools and instruments without hindering the possibility of choosing the best solutions for each kind of application domain.

Grid Architectures

Major Grid platforms like Globus [11], Legion [52], and UNICORE [53] try to provide reliable and transparent testbeds for the users to submit jobs. After an initial authentication process, users may submit their jobs to resource managers which control different hardware and software resources, across their administrative domains.

The need for standardisation has led to an ongoing effort to make Grids' distinct features compliant to Web Services resulting, for example, on the development of the *Open Grid Service Architecture (OGSA)* [48] specification. These efforts include the extension of Web Services towards Grid Services, in order to have a simplified way to both access and combine different types of Grid resources. Advances on Grid services have been extensively discussed in the *Open Grid Forum (OGF)* (former *Global Grid Forum (GGF)*) with the intent of the pervasive adoption of Grid computing both for research and industry. [17].

1.1.2 Difficulties of Grid Application Development

The difficulties of Grid application development occur at different levels, namely *application level*, *development/programming level*, and *system level* (which include middleware and system architecture layers).

Application Level Difficulties Due to the complexity of the Grid environments, an application developer will have difficulties in understanding how the logical

specification/characteristics of an application relate to the system organisation, its distributed architecture and the corresponding software and hardware resources; a suitable compromise must be sought between the required level of transparency and the degree of user control over the execution environment. Of course, this is critically dependent on user and application profiles. It may also happen that the levels of transparency and user control may be required to adapt, depending on the evolution of the computations. As an example, consider a situation where adequate Quality of Service must be satisfied by the system: if the parameters defining the quality of service reach unacceptable values, the user may want to have an active role upon application (and system) reconfiguration.

It is also difficult to understand how computation and data access application characteristics may affect the efficient usage of the allocated Grid resources, thus making it extremely difficult or even impossible for the user or application developer to make decisions concerning the appropriate mapping between the needed and available resources.

The above difficulties can be overcome by providing:

1. adequate development/programming environments;
2. adequate middleware/system support that contribute to ease the mappings from the logical application characteristics to the allocated system resources, and also that allow their dynamic reconfiguration.

Development/Programming Level Difficulties At this level, the main difficulties are due to the complexity of applications (in science and engineering, but also in business) built out of a large diversity of heterogeneous components (some of which can be legacy codes of high internal complexity), which are based on different programming and computation models, and that may require distinct (and sometimes incompatible) execution support environments.

Due to the above, adequate abstractions should be provided in order to support clear separation of concerns, in the following dimensions:

- regarding the logical component specification and its execution environment, allowing a clear separation between the logical application organisation and its system level deployment;
- regarding the component individual interfaces and how they are interconnected in order to build a global application structure;
- regarding individual component behaviour and how global application components are coordinated, possibly including reconfiguration and adaptation to change;

System Level Difficulties The difficulties at system level, including middleware and system architecture layers, are related to the issues of how distributed operating systems for Grid platforms will be capable of handling scalability (physical, number of users, hardware/software resources), heterogeneity (in computation, storage and communication physical resources, and also in the logical or software resources), dynamic nature (in terms of failure; of unpredictable variation in system behaviour; and of modifications in hardware/software components and services), and the span of system administrative domains, coupled with the critical issue of security.

In most existing Grid systems, the user interface is too low-level and mostly dedicated to job submission. It is still assumed that the application developers should have a solid knowledge of the interface details for resource allocation, and their proper orchestration with data location and file management. Consequently, adequate facilities for resource composition and coordination are still lacking in those systems.

In order to solve such difficulties, the integration of high-level abstractions, for example based on components and workflow management tools into Grid environments has proven extremely useful for simplifying Grid application development. *Component-based models* encapsulate different kinds of resources and with different granularities, thus providing a clearer and simpler interface for their access. Nevertheless, the composition of components supporting adequate data and control flows is a difficult task, moreover considering the large-scale, dynamic, and heterogeneous characteristics of Grid environments.

Considering such difficulties, *workflow systems* for Grid environments aim at improving application development support (e.g. [4, 23, 206]). Concerning its structuring and composition, workflow systems based on components support specific data and control flow mechanisms for defining data paths and enacting component execution. Usually a straightforward Graphical User Interface for component composition is available, but this is not mandatory, an alternative being a textual workflow language. In the lower layers, support for the workflow execution exist.

Workflow systems enable important functionalities:

- component reuse and composition;
- adequate User Interfaces (UI) for application specification;
- adequate interfaces and mechanisms for their integration into the enclosing environment, flexibility to incorporate script-based control languages, and flexible interfaces to the underlying resource management layers and execution support systems, including the Grid;
- managing the entire life cycle of application, including specification, deployment and execution, and dynamic reconfiguration.

1.1.3 Problem Solving Environments (PSEs)

In our work, we are interested in approaches based on *Problem Solving Environments (PSEs)*. PSEs are integrated environments which help scientists and engineers to solve problems in their specific domains. *PSEs* integrate specific models (like models for representing the human body, or models to represent the wind flows on the atmosphere), and *generic or specific tools* to evaluate or control those models. An example of a generic tool is a 3D visualization front-end which may be used in both a medical *PSE* and a weather prediction *PSE*. However, a medical *PSE* may integrate a specific tool like a controller for a medical robot, whereas a weather forecast *PSE* may integrate a different specific tool like a mathematical engine for wind and temperature analysis.

This is an area of intensive research, requiring expertise from very different domains in science and engineering, fitting what is commonly designated as computational science and engineering [90,91]. PSEs are particularly helpful for complex applications where large number of users may interact at an abstract level, namely using the languages and models of the specific scientific domains, as well as adequate user interfaces. Hence, *PSEs* require support platforms capable of combining heterogeneous distributed computational components, and of transparently supporting the complex interactions between the components and the users.

Traditionally, PSEs have been providing specific support to the development and execution of experiments in science and engineering. In general, the user interface is a virtual workbench that tries to simulate a real laboratory and which is accessed using a high-level language, specific to the problem domain.

The main goal of the PSE developers is to combine the best of two worlds: to provide transparent access to the specific software resources required by applications, and to better exploit the available distributed computational capabilities. Since a large number of traditional PSEs for different application areas include common resources or tools with similar functionalities, *middleware* platforms try to capture and provide those similarities.

Middleware platforms for PSEs [90, 112–114] offer the necessary generic support to build application specific PSEs, thus simplifying the developers' tasks. Those platforms provide facilities for a broad class of applications in the areas of Environmental Engineering, Weather control, Chemistry and Physics Engineering, Mechanical Engineering, among others.

Furthermore, those generic platforms follow the principles of the *component paradigm* [27, 100], what is important for reusing and composing the resources to assemble a new particular PSE. An important evolution of this component structuring is the need to add support for dynamic operation, i.e. PSEs with configurations that may evolve in time and allowing users to control those changes directly.

In the above we explained the reasons for our focus on PSEs, particularly, PSEs which are amenable to being structured as collections of cooperating components. In

Environments	Problems
PSE	High performance requirements Massive data processing Highly heterogeneous tools Flow dependencies Reliability Security
Complex interactive environments	Tool synchronisation Consistency Cooperative work
Dynamic environments	Dynamic reconfiguration

Table 1.1: *Issues addressed by PSEs.*

the following we discuss a number of issues addressed by PSEs as illustrated in table 1.1:

- Many scientific applications usually have high-performance requirements which use complex mathematical models requiring the support of parallel processing systems.
- Many applications produce large amounts of data. As such, PSEs have to give support to the storing, management, and transmission of huge bulks of data.
- The computational components are heterogeneous, both in their hardware and software requirements, as well as in their computational models (e.g. sequential, concurrent, event-driven).
- In other situations, e.g. like code coupling simulations, *PSEs* may have to integrate models from different scientific areas (like mathematics, physics, biology, geography), so that expert users from different areas may communicate, or even use tools from another area without having to know the intricacies of the scientific language of that area.
- In an integrated *PSE*, tools may exhibit mutual dependencies. For example, if the execution of two cooperating tools involves frequent interactions (e.g. on the throughput of the flow of data), their mappings to real processors should take such dependencies into account.
- Reliability may also be mandatory. Several applications are critical systems (e.g. medical systems) which require continuous execution with fault-tolerance.
- Many distributed *PSEs* have to address security issues.
- Additionally, *PSEs* show characteristics common to complex interactive environments. Such kind of problems are a consequence of the user capabilities that the underlying execution system will have to guarantee. For example, an application

may critically depend on the correct synchronisation of a set of heterogeneous tools, each one showing specific hardware and performance requirements.

Considering another example, a set of users may want to cooperate on the steering of a scientific experiment (i.e. control of the parameters of the experiment at run-time, so that different execution scenarios can be analysed and compared). The system will have to support some coordination, so that parameter steering by the users is consistently applied and perceived by the concurrent users (observers).

- Finally, future *PSEs* aim at solving additional kinds of problems not present in current *PSEs*, such as problems related to dynamic reconfiguration and adaptability. For a comprehensive discussion on future trends of *PSEs*, please refer to [38].

The Cycle of Activities of a Typical Problem Solving Environment

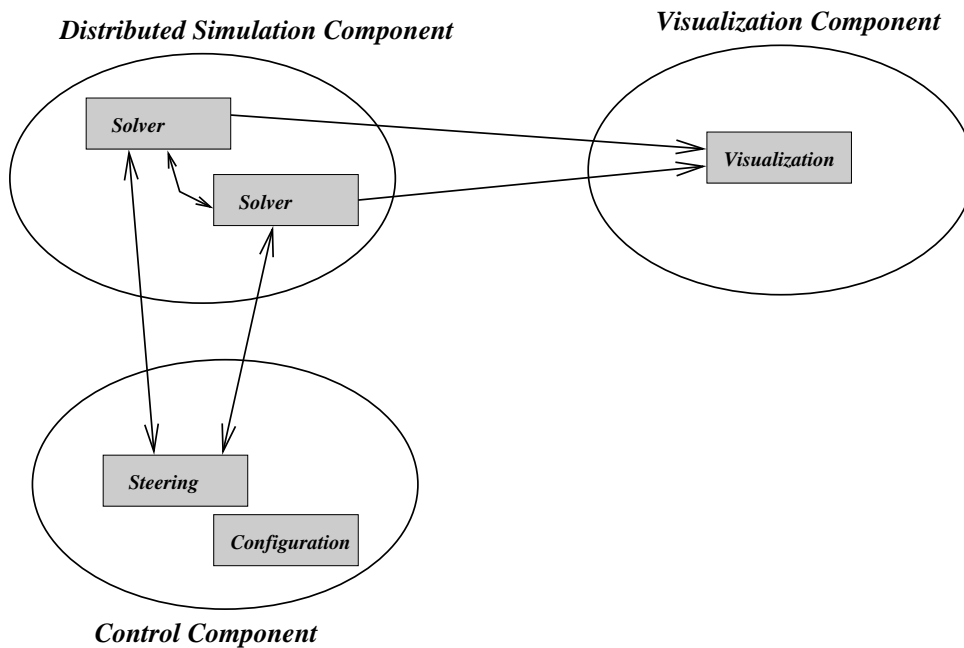


Figure 1.1: A typical logical architecture for a distributed/parallel *PSE* consisting of three components – a distributed simulator, a visualisation and a control component.

Typically, the distributed architecture which underlies a *PSE* is composed of the following main elements (see Figure 1.1): application components (e.g. for simulation) which may be run in parallel or in a distributed platform; visualisation components; and control components (e.g. for steering). When implemented with associated support tools for user interaction and assistance, the *PSE* provides a complete environment to support the user, throughout all application development and execution phases. These phases may be represented through the following sequence of activities [36]:

1. Problem specification using an application specific model (for instance, an algorithm for parallel and distributed simulation).
2. Configuration of the logical architecture of the PSE, achieved by component selection (for example, components that represent the simulation, components for visualisation, and components for execution control), and their associated support tools.
3. Component activation and mapping onto an underlying architecture.
4. Initial definition and setup of the parameters of the application components, depending on the selected type of the application model (for example, a simulation may be executed under different models, each one needing specific parameters).
5. Start of the execution with specification of observation and control functionalities (for example, start of the simulation, with monitoring and steering).
6. Interactive control of the execution.
7. Analysis of the intermediate or final results.

The above steps may be repeated cyclically until the desired final results are obtained. Depending on the specified modes of operation, the final results may have to be logged into files for post-mortem processing or passed to other tools or subsystems. For example, in the simulation example, in steering mode, the intermediate results are displayed on-line, and the user can dynamically modify the simulation parameters. In general, the experimentation process may lead the application developer to go back to step 1 and repeat the above cycle with different approaches for problem specification for each step.

In order to ease the above identified difficulties at the application level (section 1.1.2), long-term efforts have been trying to improve the functionalities offered by *PSEs* [18,93]. Current state-of-the-art *PSEs* are complete, integrated computing environments for composing, compiling, and running applications in a specific application domain, trying to provide all the computational facilities necessary to solve a target class of problems.

We believe that approaches based on *PSEs* do provide adequate solutions to meet the above difficulties, as mentioned at application level, as they encapsulate knowledge and state-of-art algorithms relevant to a specific application domain. However, for enabling flexible *PSE* development in new and emerging application areas, “more generic” development environments are required, that should provide abstractions and tools for component specification, programming, composition and interconnection, as well as their instantiation and deployment, via appropriate interfacing to underlying resource management systems. Such “more generic” facilities are quite helpful to build *PSEs* for specific applications.

For example, following these ideas, several ongoing projects have been trying to promote high-level paradigms for application development, based on the *workflow* and the *component* concepts [1, 15–17]. Component based models provide an effective way to develop applications from a range of different software libraries, and possibly wrapped legacy codes. Components can vary in complexity and granularity – ranging from complete applications to specialised sub-routines. The associate environments provide interfaces to specify the manipulation of components, e.g. the selection of components from a repository and their combination through a visual editor.

Features Lacking in PSEs, even for Grid Development

Some PSEs already support the deployment of applications into distributed Grid platforms (e.g. Triana [201]). However, although the user is already able to run component-based applications in a Grid, the support of structured and systematic ways of reusing components is still limited. Most state-of-the-art component-based PSEs support limited structured component composition, and have limitations regarding the support for significant changes in the structure and flow dependencies.

Many PSEs allow the user to specify direct connections between components, for example through channels and ports, but usually still lack full support on important aspects like:

- explicit support for defining and composing new (typical) structures for component interconnection that may be subsequently reused in similar problems. Those (reusable) structures should be manipulable from component repositories with operations like save, recovering, and searching;
- facilities for manipulation of such structures and topologies as templates (partially instantiated). Those templates should be able to be refined and instantiated (either at development and execution times), by applying specific commands or operators;
- existence of adequate, flexible, and architecture neutral interfacing to the resource management layers of a Grid architecture, in order to support the deployment, execution, and (dynamic) reconfiguration of applications based on such typical structures and topologies.

1.2 The Need of High-Level Abstractions for Grid Application Development

1.2.1 Components and Services

Due to the complexity of Grid environments, several projects have been developing programming models based on encapsulated units such as components and Web/Grid services [20, 59, 63–67, 114]. Although such models do simplify the development process by providing units that can be composed and reused, the management of dependencies and coordination between those units is still a difficult task. Moreover, it is desirable to reuse useful components' interdependencies as a way to support less experienced users and to improve the development process. Additionally, Grid applications' execution control and dynamic reconfiguration are still open research subjects.

1.2.2 Skeletons and Design Patterns

Similarly to what has happened to general purpose programming languages and models, distributed Grid application development might benefit from the manipulation of higher-level abstractions, namely design patterns [9], as first class entities. Currently, patterns are not just a modeling abstraction anymore, but have also been included into development tools and in languages, as first class entities [61, 62].

A *Pattern* encodes a commonly recurring theme in service or component composition. It allows good practice to be identified, and shared across application domains. A pattern is generally defined in an application independent manner, and used to encode characteristic useful behaviours. Patterns are particularly useful for configuring and specifying systems that are composed of independent sub-systems. Patterns are aimed at capturing some common and generic attributes of a system – which may be further refined (eventually) to lead to an implementation.

The above concepts can meet important requirements for Grid applications, which generally need to operate in dynamic environments. Furthermore, users of a Grid infrastructure usually have different abilities, and less experienced users may find it difficult to identify useful architectural models for interconnecting components, or adequate coordination behaviours. As such, the availability of recurring patterns allows the selection of the most adequate solutions, potentially reducing both applications' development complexity and effort. Moreover, the introduction of patterns as first class entities allows the manipulation of a pattern (and its elements) as a single entity from design time to execution time, increasing re-usability and maintenance. Accordingly, pattern-based concepts may become units of both execution control and dynamic configuration.

Some component tools already provide patterns as first class entities, where pat-

terns may be defined, stored and reused independently of the individual components. Tools like the ones mentioned in [54, 55] provide a pattern-based approach for component composition – for example, the *ObjectAssembler* [55] visual development environment provides a catalogue of patterns for connecting JavaBeans components [104]. Similarly, the *Pacosuite* [54] tool supports component composition through *composition patterns* which define component interactions. Nevertheless, the patterns in those systems are neither manipulable as execution units nor dynamic reconfiguration units.

The Grid community has already recognised the importance of patterns [47] as a way to re-use expert knowledge, but those still have limitations and are generally still not available as a programming paradigm for the Grid (or integrated in Grid software development environments). Works on *skeletons* for Grid computing [195, 196] represent a related approach towards reusability of expertise within Grid Environments. Specifically, *skeletons* are programming abstractions (most often inherited from functional programming and parallel-processing systems) that are sometimes amenable to optimised implementations of typical parallel algorithms. Nevertheless, the available skeletons present on those Grid Environments are not kept as first-order abstractions throughout a Grid application’s life cycle.

1.2.3 The Main Goal of this Work

Motivated by the above considerations, we argue in favor of an approach which aims at providing patterns to Grid environments in two main dimensions, namely structural and behavioural. Such separation of concerns contribute to increase flexibility on pattern-based application configuration.

The aim of this work is to contribute to simplifying the development of distributed applications, namely mapped to Grid environments, by providing a novel way to compose and manipulate their components. Specifically, the goal is to enhance the application development cycle supported by Problem Solving Environments by providing:

- Reuse of typical application configurations.
- Structured support for constructing new configurations and controlling their execution.
- A systematic methodology applicable in all the stages of the application development cycle, from application specification to execution and reconfiguration control.

Our work methodology was based on observing typical PSEs and Grid applications, where common and recurrent interrelations/associations emerge, both at the structural level (e.g. direct connections between the PSEs’ components and, for example, common software architectures in high-performance computing applications) and

at the behavioural/coordination level. The focus was then to try to capture those common interaction patterns into well identified abstractions. These are made available for reuse through an uniform and extensible model which allows the user to combine and control those abstractions in a structured and systematic way. A first overview of our approach, which we designate as *model*, is described in the following section.

1.3 The Proposed Model

The *model* presented in this dissertation elects *patterns* as the main abstractions that are kept as *first class entities* during the entire application development cycle. In this way, patterns can be manipulated either at design, execution, and reconfiguration times. That is, patterns are manipulable units for configuration, execution control and reconfiguration actions.

In the model, application configurations result from the composition of patterns (e.g forming different topologies and hierarchies) and may also be changed through pattern replacement or refinement. During execution time, individual control of patterns is also possible. Moreover, and with proper execution control, dynamic reconfiguration may also be achieved. All these actions for pattern manipulation are performed through a variety of *Operators* both for design and execution times where all operators act upon patterns in a uniform way in all phases of the development process and for the different manipulable patterns.

Patterns and operators are used to support the specification and manipulation of the application configuration as composition of patterns, and these can be individually manipulated through adequate operators. Additionally, the distinction between two main categories, namely structural and behavioural, both for patterns and operators, provides the model with flexibility on application configuration, reconfiguration, and control. A pattern from one category (structural or behavioural) may be combined with different patterns from the other category, and also structural patterns may be manipulated independently from the associated behavioural patterns, and vice-versa. This is quite important for application development, as shown in Chapter 7 concerning the assessment of the model.

Furthermore, the persistence of patterns and their manipulation through operators throughout the application development cycle promotes an uniform view of the model. Finally, the model promotes a methodology which on one hand may guide the user, and on the other hand may be (totally or partially) automated through scripts.

The following sub-sections describe structural and behavioural patterns and the methodology proposed.

1.3.1 Structural and Behavioural Patterns

Structural Patterns capture component connectivity and represent common ways of

combining components within a given application domain, as well as of reusing them across several applications. Examples of Structural Patterns are:

Pipeline: e.g. the detection of waves in a Cosmology application as the first stage, its analysis/processing through subsequent stages, and visualisation of results in the final stage;

Star: commonly underlying the Master/Slave processing in parallel applications where the nucleus, i.e. the Master, divides a problem into independent sub-problems and sends these requests to be processed by the satellites, i.e. the slaves;

Facade: common in portal technologies for Web Services, which hides the access to several distinct services (or tools, instruments, etc.) under a simplified interface.

Proxy: which usually supports the access to Grid services (through a proxy or gate-keeper).

Structural Patterns are represented in our model through *Templates* whose elements can be instantiated to executable entities (e.g tools, services, etc.) or to other Structural Patterns.

Behavioural Patterns, in turn, define interaction constraints, namely they rule the data and control flow dependencies among a set of components. Examples of Behavioural Patterns are:

Client/Server: commonly present in distributed services;

Producer/Consumer: common in workflow systems;

Itinerary: used in mobile Agent systems, where an Agent moves in order to accomplish a task, leaving behind a chain of forward pointers keeping track of the Agent's location.

Provided the above two categories of design patterns, configurations are built by selecting the Structural Patterns that best represent the connectivity between the applications' elements, and by applying upon those the appropriate Behavioural Patterns. The most adequate combinations of Behavioural and Structural patterns result from the users' knowledge on the application needs and on the capabilities of the underlying support infrastructure.

Examples of Patterns in One Particular PSE

To illustrate the usefulness of Structural and Behavioural Patterns, this section presents possible identifiable patterns in the context of the configuration of a Problem Solving Environment (this example was introduced in [44,204]). Please note that details and definitions about the patterns are given in the corresponding chapters; the idea here is to give a glimpse of the possibilities of the model.

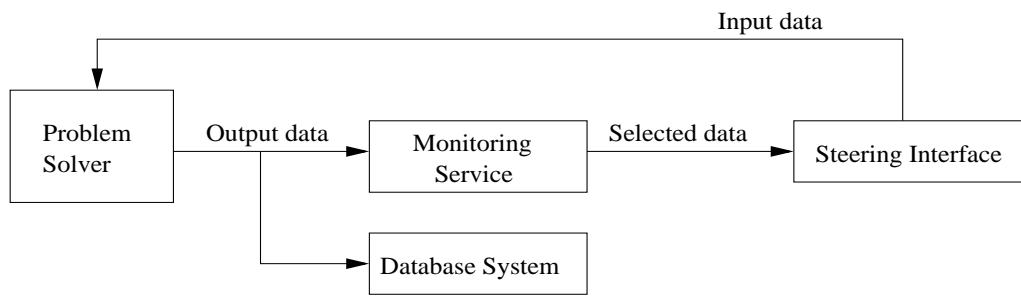


Figure 1.2: A generic PSE.

As illustrated in Figure 1.2, basic components of this PSE are a *Problem Solver* (e.g. a scientific tool) that generates data to both a *Database* and a *Monitoring Service* for data storage and filtering, respectively. Moreover, the data compiled by the latter service is fed into a *Steering Interface* which shows relevant data to users interested in controlling some parameters of the *Problem Solver*.

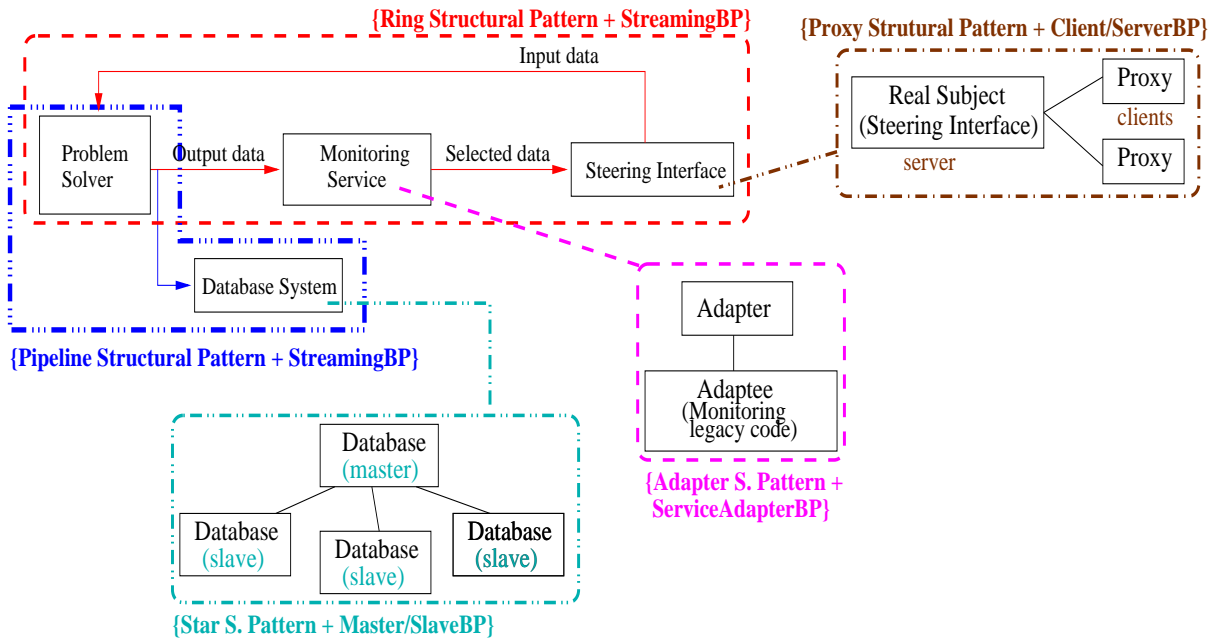


Figure 1.3: Example of the identification of Structural and Behavioural Patterns in a PSE

Figure 1.3 presents several identifiable configurations and how they are supported by corresponding *Structural Patterns*. The principal ones are:

- a) a *Ring*, establishing the necessary connections between the *Problem Solver*, the *Monitoring Service*, the *Steering Interface*, and this one back to the *Problem Solver*;
- b) a two-stage *Pipeline* connecting the *Problem Solver* and the *Database System*.

Concerning Behavioural Patterns, an adequate pattern for both structures, i.e a) and b), might be a *Streaming* Behavioural Pattern where the destinations process the continuous flow of data generated by the *Problem Solver*. This is also represented in Figure 1.3.

PSEs like this became a useful support tool within Grid environments, where the described components may represent simple or complex distributed services. For example, considering the case that the *Problem Solver* generates large amounts of data, the *Database System* component in Figure 1.2 may in fact represent a distributed storage system. In this case, a suitable configuration for this system could be a *Star* Structural Pattern (see Figure 1.3) where the nucleus behaves as the Master that coordinates storage access among a set of slave Databases organised as the star's satellites.

The *Monitoring Service*, in turn, can be provided by a legacy tool that can be accessed through the association of the *Adapter* Structural Pattern with the *Service Adapter* Behavioural Pattern (to support its access as a service). Finally, considering that several users may be allowed to cooperate on the steering of the *Problem Solver* in Figure 1.2, the *Steering Interface* might be supported by the *Proxy* Structural Pattern and the *Client/Server* Behavioural Pattern. Each user (client) would access its proxy to submit requests to a coordinator Steering Interface acting as a server. Both configurations are also shown in Figure 1.3.

Flexibility on Application Configuration

The separation of concerns related to structure and behaviour introduces a *level of flexibility* on configuring applications. The reasons are:

- a) the same underlying structure, i.e. Structural Pattern, may be combined with two different behaviours at different times;
- b) the same Behavioural Pattern is applicable to different Structural Patterns.

Such separation of concerns allows users to choose the most appropriate combinations, thereby increasing reusability of both structural and behavioural patterns.

As an example of a), one may think of a *Pipeline* Structural Pattern whose components forming the pipeline stages may interact at different times according to different Behavioural Patterns. For instance, at one time, such stages may be coordinated through the *Streaming* Behavioural Pattern, where data is automatically fed to the following stage in the pipeline. At a later time, those components may interact according to the *Client/Server* model where one stage requests data from the previous stage. In both cases, the flow of data remains the same. However, whereas in the first case the arrival of data may control the execution, in the second case, execution controls the time when data is sent.

The case b), i.e. the same type of behaviour is ruling different structures, is very common. For example, the *Client/Server* Behavioural Pattern may enforce data and control flows on diverse Structural Patterns, e.g. *Proxy* (e.g. as the *Steering Interface* in Figure 1.3), *Star*, *Ring*, or *Facade*. Likewise, the *Master/Slave* Behavioural Pattern may be useful, for example, on the *Star* Pattern (represented in Figure 1.3), as well on a *Facade*

Structural Pattern that may interface to dissimilar, although functionally equivalent, sub-systems.

Another *level of flexibility* results from the possibility of combining Structural Patterns and associated Behavioural Patterns into *hierarchies*. As such, a pattern may contain elements which are patterns themselves. For example, a stage in a *Pipeline* may be itself an *Adapter Pattern*. The behaviour of the elements of the outer pattern (i.e. the *Pipeline*) may be ruled by the *Producer/Consumer* Behavioural Pattern, whereas the elements in the inner pattern (i.e. the *Adapter*) may behave according to the *Client/Server* Pattern. Therefore, this resulting *Hierarchical pattern* is coordinated differently at each individual pattern that composes it. Furthermore, inner patterns are still directly manipulable.

Such hierarchical structuring on pattern composition, supports the commonly dual approach towards configuration, namely either top to bottom or vice-versa. On one hand, the user may start by selecting the most adequate patterns that best represent the overall application's architecture, and only then define and embed into those patterns, the appropriate patterns for the sub-elements in that architecture. On the other hand, the user may start by identifying all the necessary elements needed for the application and their associated patterns, and only then aggregate those patterns into higher-level configurations.

In the next section we discuss how Structural and Behavioural Patterns may be manipulated through Structural and Behavioural Operators to support a finer application configuration and execution control. A reconfiguration example supported by operators is also presented in the section.

1.3.2 Pattern Operators

Pattern Operators are the abstractions that manipulate patterns and in such a way that the patterns' intrinsic characteristics are preserved along all the phases where operators are applied. In this way, besides persisting from design to execution time, patterns are handled in a uniform way throughout the application life cycle.

Pattern operators may be applied in pre-defined ordered combination and may be shared between users. These operator sequences may be inserted into scripts providing automated configuration, reconfiguration, and execution control.

Operators are also divided into Structural and Behavioural. *Structural Operators* enable a constrained way to modify Structural Patterns, i.e. the distinctive configuration of each of the manipulated Structural Patterns remains the same. A list of the Operators available is given in a section ahead, but some examples of pattern refinement through Operator application are:

- deleting one of the satellites of a *Star* Pattern by applying the *Decrease* operator;
- adding another sub-system to a *Facade* Pattern through the *Increase* operator;

- or using the *Embed* operator to insert a Structural Pattern into another Structural Pattern, forming a hierarchy (e.g. such that the nucleus of a *Star* Pattern template becomes in fact a Pipeline Pattern template).

Behavioural Operators act upon the final application configuration, which comprises Structural and Behavioural Patterns, for ruling data and control flow dependencies. Some examples are:

- to *Repeat* the execution of a pattern a certain number of times, e.g. to run twice a pipeline whose elements are coordinated by the *Streaming* Behavioural Pattern;
- to *Limit* the execution of a pattern, i.e. such that in the case the execution does not terminate within a pre-defined time, the execution is aborted;
- to *Stop* (i.e. suspend) and *Resume* the execution of a pattern.

Behavioural operators support abstractions for application execution control, and for static or dynamic reconfigurations. Sequences of applied behavioural operators can be incorporated into scripts and reused for replaying a specific execution history.

The next subsection describes the combined application of some Structural and Behavioural Operators in the case of the PSE example described in sub-section 1.3.1.

Configuring and Controlling One Particular PSE

After having identified the useful Patterns for the PSE in Figure 1.3 presented earlier, the user selects them from a repository and combines them through some Structural Operators. The components within each resulting Structural Pattern (SP) are connected according to its definition. These SP are in fact pattern templates whose elements must subsequently be instantiated to tools/services by the user.

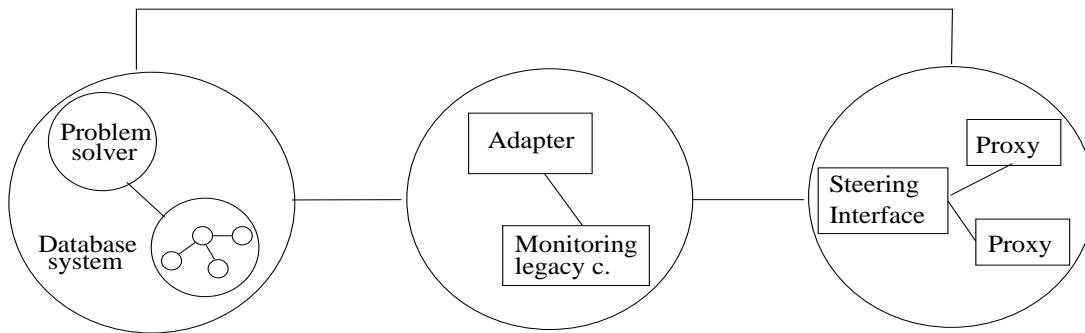


Figure 1.4: *Pattern-based configuration of the example in Figure 1.3. All elements of the Structural Patterns are already instantiated to the necessary tools/services.*

Figure 1.4 shows the final structural configuration for that PSE, where the necessary number of elements for each Structural Pattern may be defined through the *Increase* and *Decrease* operators.

Pattern hierarchy is defined with the *Embed* operator: a Star SP representing the *Database System* was embedded in the second stage of a Pipeline SP representing the connection to the *Problem Solver*; this Pipeline SP was in turn embedded in one stage of a Ring SP; the other two subsequent stages were defined by embedding, respectively, an Adapter SP supporting the *Monitoring Service* and a Proxy SP for the *Steering Interface*.

In the following we present a possible Structural Operator sequence to build the configuration in Figure 1.4. Please note that this operator sequence only generates the necessary patterns and combines them, i.e. the instantiation of the pattern elements to specific tools/services (e.g. the *Problem Solver*, etc.) is absent in the sequence:

```

1: Create( RingSP, ``PSE``, 3 )
2: Create( PipelineSP, ``DataStoring``, 2 )
3: Create( StarSP, ``DatabaseSystem``, 4 )
4: Embed( DatabaseSystem, DataStoring, ``cph2`` )
5: Embed( DataStoring, PSE, ``cph1`` )
6: Create( AdapterSP, ``MonitoringSv`` )
7: Create( ProxySP, ``SteeringInt`` )
8: Increase( 1, SteeringInt )
9: Embed( MonitoringSv, PSE, ``cph2`` )
10: Embed( SteeringInt, PSE, ``cph3`` )

```

Please see the following chapters for the detailed description of the above operator sequence.

Finally, the PSE's configuration is completed: a) by selecting the necessary Behavioural Patterns referenced earlier and associating them to the Structural Patterns; and b) by instantiating all the elements (i.e. component place-holders) in the templates.

Subsequently, the user may launch the execution of the application, and control it through Behavioural Operators. For example, by applying the *Restart* operator to the (outer) Ring pattern in Figure 1.5 will (recursively) launch the execution of all the components in a periodic way. One argument of *Restart* defines the period of time when the application's execution should be automatically restarted. At a later moment, the application of the *Terminate* operator will cease the current execution (in case there is one) and the invocation of the *TerminateRestart* operator will abort the defined automatic restart.

The user may also apply Structural and Behavioural Operators to reconfigure the PSE in Figure 1.5. Namely, Figure 1.6 presents the configuration of the PSE after:

- a) The *Monitoring Service* is replaced with a pattern representing a more sophisticated service. This is the result of the application of the *Replace* Structural Operator.
- b) The usage of the *Increase* Structural Operator to add a new client of the *Steering Interface*. This reconfiguration may be performed at run-time.

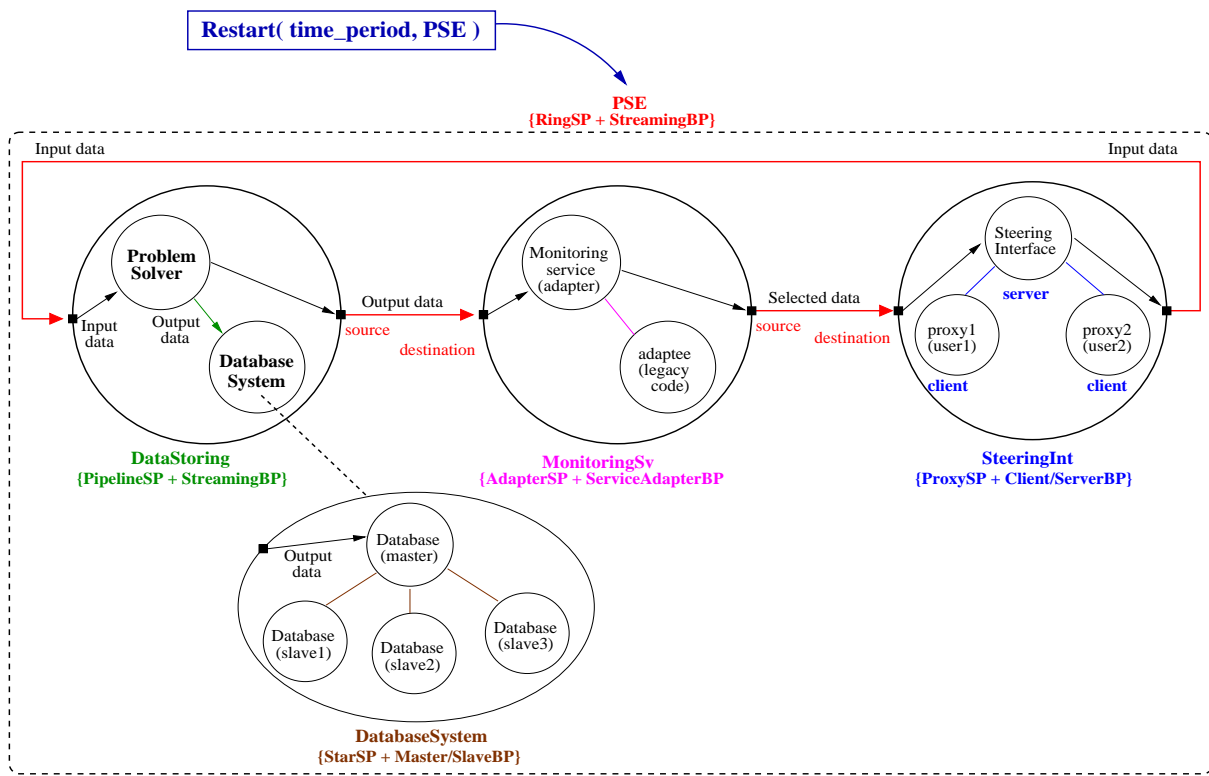


Figure 1.5: *Launching periodically the execution of the PSE.*

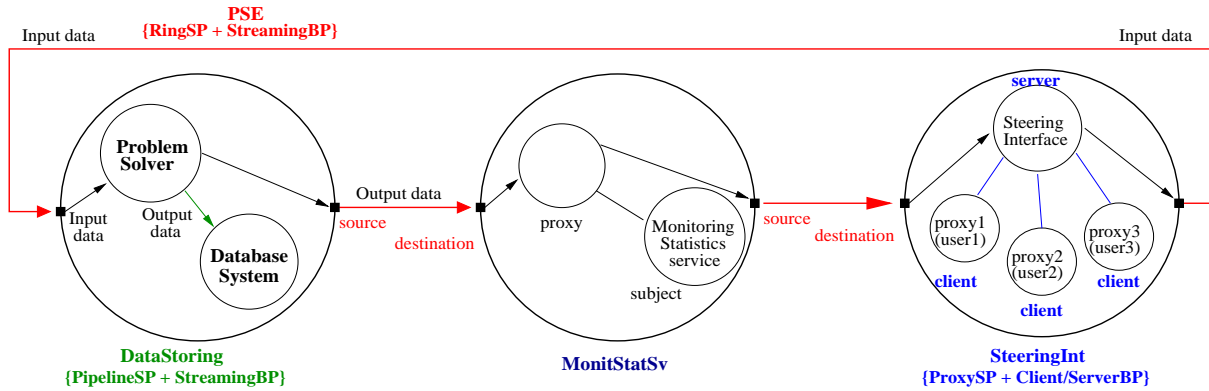


Figure 1.6: *New configuration of the PSE (e.g. providing support for an additional user of the Steering Interface, i.e. "proxy3").*

1.3.3 A Methodology within the Model

To summarise the above descriptions, the model is applicable at three different stages, namely at configuration, at execution, and at reconfiguration times:

- At configuration time, component connectivity is defined through Structural Patterns. Structural Pattern Templates can be composed and refined through Structural Operators.
- At execution time, the user can control application execution through Behavioural Operators. The executable application results from instantiating the templates with executable components. Behavioural Operators act upon the combined Behavioural and Structural Patterns, for example allowing the user to stop and resume execution of the patterns, or

to launch them periodically.

- At reconfiguration time, Behavioural Operators support reconfiguration in the dimensions of structure and behaviour, including one independent from the other.

Both the creation of pattern instances and their manipulation through operators may be defined through scripts for systematic usage.

The above actions define a *methodology* within the model. On one hand, such methodology may guide a less experienced user on programming and controlling an application based on patterns. On the other hand, the methodology may also define a systematic approach for both more and less experienced users as it may be automated into scripts.

In general, the methodology is based on the following stages:

1. Structure definition by selection of Structural Patterns and their refinement through Structural Operators.
2. Behaviour definition (i.e. definition of data and control flows) by selection of Behavioural Patterns which are associated to the selected Structural Patterns.
3. Execution control based on Behavioural Operators.
4. Dynamic reconfigurations supported by Behavioural and Structural operators.

Upon execution finalisation, or when execution abortion is explicitly requested by the user, steps (1) and (2) above may be repeated, and also intertwined, defining a development time reconfiguration.


1.3.4 Assisting Application Development in PSEs

This section discusses how the full implementation of the proposed model assists application development. Specifically, a mapping between stages in the presented model and the application development life cycle illustrates the adequacy of the model to that purpose.

Using the Model to Enhance the Application Development Life Cycle

The relevance of the model composed of patterns and operators may be analysed in the context of a Problem Solving Environments' life cycle. The different stages needed by our model are directly mapped to the PSE life cycle, although some extra requirements are necessary, as presented in Figure 1.7.

- In the *first step* (1 in Figure 1.7), besides the definition of the application model, it is necessary to perform an evaluation of the most appropriate configurations and the required interactions between the elements of the model. As such, the user is required to have some knowledge (and/or advice) of the necessary state-of-the-art configurations and behaviours of the application. This may (or not) be already provided by our approach, as well as how to use those configurations/behaviours for building specific configurations that suit the application model the most.



1	Problem specification.	Problem specification. Configuration evaluation.
2	Configuration: component selection.	Configuration: component and pattern selection. Refinement through structural operators.
3	Mapping onto the underlying architecture.	Mapping dependent on the underlying resource manager.
4	Parameter setup.	Parameter setup.
5	Start of the execution. Evaluation and change of the application status.	Application of the operator "Start". Monitoring and Steering components are necessary.
6	Control of the execution.	Application of behavioural operators.
7	Analysis of results.	Application of a log-related operator for analysis of the pattern-based application. A log component is needed.

Figure 1.7: *The software life-cycle of application development in PSEs (left column) and the mapping of the application of the pattern and operators model to that life-cycle (right column).*

- In the *second step* (2), the user selects the adequate Structural Patterns, instantiates them with the necessary components, and defines their flow interdependencies by selecting and applying the adequate Behavioural Patterns. The structural operators allow the definition of the configuration in an incremental way.
- The *third step* (3) is dependent on the specific implementation of our model on a given architecture. In the prototype implementation of our model over the Triana environment (Chapter 6), the user may select for example, which components are to be executed remotely or in parallel. Considering a different underlying system, such decision may be transparently made by some support tool.
- The *fourth step* (4) remains the same.
- In the *fifth step* (5), the user launches the execution using the behavioural operator *Start*. Nevertheless, the requests for observation and modification of the application status are dependent on the existence of monitoring and steering components tailored to the application model.
- In the *sixth step* (6), the user is supplied with different behavioural operators which allow for instance, to suspend application execution and to resume it afterwards, or to define how many times the execution will be repeated.
- In the *seventh step* (7) the user may apply, for example, a log-related behavioural operator

(in order to produce a trace of the execution history) and inspect data associated with a given pattern-based configuration.

Finally, the cyclic repetition of the above steps represented in Figure 1.7 may be mapped to the discussed reconfiguration dimensions of the proposed Pattern- and Operator-based model.

1.4 Contributions of the Thesis

In the previous section, we gave an overview of the model and the methodology for developing applications targeted to Grid environments and this presentation was illustrated with an example of development of a PSE. The main contribution of this thesis is the proposal of an approach providing Structural and Behavioural/Coordination Patterns and Operators. There is also an associated methodology which may guide the user on structured and systematic application construction. We designate the proposed approach as *“a model for pattern- operator-based application development”*. The defined model aims to contribute to the simplification of Grid programming, specifically in the context of Problem Solving Environments.

1.4.1 Work Approach

The phases of our work which led to this dissertation were the following:

1. Evaluation of the state-of-the-art concerning Problem Solving and Grid Environments, as well as abstractions for distributed application development and execution control.
2. Proposal of a Pattern and Operator-based model providing abstractions for specification of structure and behaviour, execution control, and reconfiguration.
3. Investigation of how Structural and Behavioural patterns may be used to abstract typical application scenarios in Grid environments.
4. Definition of an abstract architecture supporting the model.
5. Mapping of the abstract architecture onto a Grid-aware environment – Triana, and development of an experimental prototype.
6. Evaluation of the model through selected applications, and their experimental validation using the developed prototype.

The above stages were developed incrementally and several iterations were performed.

1.5 Dissertation Outline

This dissertation contains eight chapters, whose contents are summarised below:

Chapter 2. This chapter describes state-of-the-art useful paradigms for Grid programming, and highlight the importance of (higher-level) abstractions such as skeletons and patterns to represent and reuse typical component interactions in Grid applications.

Chapter 3. This chapter discusses the characteristics of the model. Namely, the model is based on Structural Patterns and Operators, and Behavioural Patterns and Operators. The semantics of Structural and Behavioural Patterns are described in this chapter, whereas operator semantics are discussed in Chapter 4. Chapter 3 also describes the basic methodology associated to the model, whereas an extension to the methodology and the reconfiguration capabilities of the model are discussed in Chapter 5.

Chapter 4. This chapter describes the semantics of the Structural and Behavioural Operators.

Chapter 5. This chapter describes the capabilities of the model towards structured reconfiguration based on pattern manipulation, both on development and execution times. The model starts describing an extension to the methodology discussed in Chapter 3 for pattern manipulation through the application development cycle, followed by a discussion on the possible application reconfiguration strategies as a result of pattern manipulation through operators.

Chapter 6. This chapter illustrates the partial implementation of the proposed model over Triana, a Grid-aware and workflow-based Problem Solving Environment. The Chapter also discusses a possible mapping of a small sub-set of the behavioural operators to the *DR-MAA*, a distributed resource manager API for execution control.

Chapter 7. This chapter presents a set of examples highlighting the capabilities of the model, some of which are case studies based on the developed implementation of the model over the Triana workflow system.

Chapter 8. This chapter summarises the achievements of the research work described in this thesis, and lists open issues, which will ground future research work.

2

Abstractions for Grid Programming

Contents

2.1	Introduction	26
2.2	General Solutions	27
2.3	Solutions for Structure and Interaction Reusability	32
2.4	Skeleton/Pattern-based Models and Systems	39
2.5	Summary	47

This chapter discusses solutions for distributed application development based on high-level abstractions such as skeletons and design patterns and their integration in programming environments. The importance of other Software Engineering abstractions such as Component/Service systems and Dynamic Reconfiguration for Grid application development is also highlighted.

2.1 Introduction

According to Foster [191] a system may be considered a Grid if it fulfills three requirements:

- A Grid isn't subject to centralized control. A Grid provides integration and coordination for resources at different control domains, from diverse entities at a unique or across different administrative domains providing users with support on issues such as security, policy, payment, membership, etc.
- A Grid is based on standard, open, and general-purpose interfaces and protocols. These support essential issues such as authentication, authorization, resource discovery, and resource access.
- The interfaces and protocols provide some level of quality of service, in terms of security, throughput, response time, or the coordinated use of different resource types.

Moreover, the features distinguishing Grid environments from other distributed computing approaches include: *heterogeneity* and *dynamics*. Specifically, the infrastructure can change significantly over the lifetime of a single application, it is composed of a range of different platforms, and it may be managed by different administrators (see [22] for a useful survey).

Users utilising a Grid infrastructure possess very different abilities, and less experienced users may find it difficult to identify useful architectural models for interconnecting components/services. Consequently, the existence of a pre-defined set of patterns/schemes is therefore particularly useful in this context, and their relevance for Grid environments has been increasingly recognised in the community [38, 47, 143, 186]. Once components and services have been connected together, another major difficulty is the need to identify suitable coordination mechanisms between them. Providing a set of operators for execution control and orchestration jointly with the abstractions at the "behavioural" level is therefore important.

Therefore, the work in this thesis aims to extend Grid application development environments with structuring mechanisms based on commonly recurring patterns. Using a library of design templates and pattern operators, a user is able to combine these with other specialised components that may be required in a particular application domain – both at design and execution times.

The difficulties of programming in a Grid Environment have already been discussed in Chapter 1. In the following section, we introduce some paradigms from the area of software development; these concepts proved their usefulness in the software development process, and some of them are already used for Grid program development (please see [95] for an extensive study on *Grid Programming Models*). The goal of that section is to describe paradigms that had influence on the definition of our proposed approach. The subsequent sections, in turn, highlight the importance of higher-level abstractions to represent component interactions, such as skeletons and specifically patterns, to be included as constructs for application development.

2.2 General Solutions

Solutions to some of the mentioned types of problems have been included in several state-of-the-art distributed environments (applications and systems). Low-level and middle-level distributed systems like Globus [11] are allowing transparent access to distributed high-performance executing platforms, and related efforts have led to the development of several paradigms.

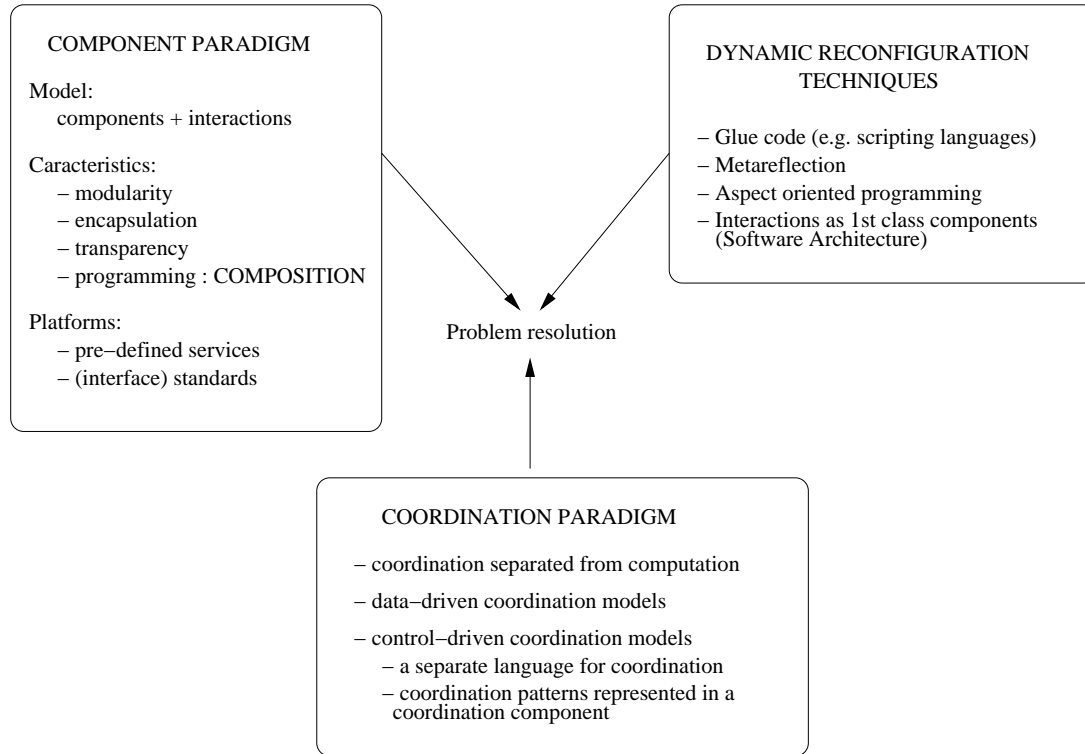


Figure 2.1: Useful paradigms and techniques and their characteristics.

Figure 2.1 identifies relevant paradigms for distributed programming, which are briefly discussed in the next subsections.

2.2.1 Component Paradigm

Firstly, one significant contribution came from the *Component* concept¹ which further simplifies the structuring of distributed applications and increases productivity.

The *Component Paradigm* [27, 100] is adequate for reuse because it extends the traditional *Distributed Object Paradigm* (e.g. underlying *JavaRMI* [107] and the original CORBA specification [105]) offering a major decoupling between the entities that build distributed applications. A component represents a self-contained abstraction that explicitly defines its functionality as well as its context dependencies through an interface specification. The specification usually follows the rules of a initially discussed *component standard* [100].

¹We assume a component definition as the one in the area of *Software Architecture* [98, 99].

The component abstraction in itself presents a larger granularity than the object abstraction, where, in fact, a component may be built out of several object implementations. These objects may be executing in a distributed environment, and together provide the complete component's functionality. Nevertheless, a component may be programmed under a different paradigm other than the object paradigm. In any case, the component has always to specify in a clear way which is its contribution and what the component expects from the executing environment. Component platforms view components in this way, and allow high-level applications to be built as structured distributed environments [104–106].

The components' characteristics of modularity jointly with high-granularity, encapsulation, transparency, and interface specification, help to reduce and clarify the dependencies of one component from other entities. Consequently, developers find it simpler to combine components and replace them.

Furthermore, there has been a proliferation of off-the-shelf components which represent complete (or partial) enterprise applications that can be transparently accessed by the users. Enterprises guarantee reliability of the component's services which the users access as simple *black-boxes*. Due to these characteristics, programming in component based distributed environments is strongly related to *composition* (e.g. [28]). End-users select the most adequate components and aggregate them through *composition techniques* which hide component heterogeneity but satisfy, at the same time, users' requirements for the final complete application. *Component platforms* [104–106] integrate several services for component management and provide the adequate environment for component composition under some *component standard*. Furthermore, the *Service-Orientated Computing paradigm*, e.g. underlying *Web Services* composition, has extended the object-oriented and component paradigms defining the composition of (loosely coupled) services [87–89].

Therefore, component and service based software development provides an effective way to develop applications from a range of different software libraries, wrapped legacy codes, and through the access (e.g. discovery and connection) to services. Such components can vary in complexity and granularity – ranging from complete applications to specialised sub-routines. “Problem Solving Environments” (PSEs) [18,90,91,94], in particular, are examples of those environments, traditionally in the areas of science and engineering, for Grid computing application development. For example, the Triana System [82] provides access to many components representing tools, but also provides access to Web Services similarly to those components. Namely, Web Services are available in Triana's tool box, and they are represented as Units in workflows in the Triana's canvas similarly as the Units representing tools.

A number of projects (see a list in [1,16]) have explored component composition and workflow management for components and services in the context of Grid computing [15,17]. Generally, these environments involve a user interface which enables components to be selected from a repository, and combined using an editor. The interfaces to the components are generally pre-defined, and often expressed in a standardised form (e.g. XML [76]).

Such environments generally consist of 3 tiers:

- a user portal to enable interaction with the components;
- a series of middle tier services – such as a data management service, one or more compute

services, etc; and

- the physical resources on which the components are to be executed.

Manipulating either individual components or groups of components is a useful extension – and a complete support to achieve this is not yet directly provided in existing environments. It is also useful to determine and abstract common interactions between components, and to make these abstractions available to a user.

One novel theme addressed in our work is the ability to view component and service composition (to solve a particular problem) as being equivalent to manipulating a Structural Pattern using pre-defined Operators. Subsequently, the resulting structure can be manipulated via Behavioural Operators that enable multiple data flows to co-exist within a system. A user (application developer) may identify useful Structural or Behavioural Patterns – in particular application contexts – and record these within a patterns library. These can then be configured using an operator library.

Modeling Interactions

A number of approaches exist already for modeling interactions between components in the context of Grid environments, or for developing formal models of job submission and management in a Grid [31]. These, however, still provide very limited support for enabling a user to subsequently utilise the outcome of these models. For example, Marinescu [3,4] provides a common abstraction for modeling workflow to support Web and Grid Services. The approach is centered on developing graphical abstractions that can be used to model interaction patterns between components. The graphical patterns model aspects such as AND/OR/XOR based interactions – and the focus is to support a workflow enactment engine that may be used to co-ordinate component execution.

Similarly, a key emphasis in the Fraunhofer Resource Grid [2] is on developing a Grid Resource and Job definition language, to enable job submission, resource selection, and allow a description of dependencies which exist between resources. In this work, the Grid Job Definition Language may be mapped to a series of parameterised Petri Net (PN) blocks. Each block represents some aspect of the language such as Task execution and synchronisation, Conditionals and Choice, and loops (such as the `While...do` loop etc). Each PN block is encoded in XML based on the Petri Net Markup Language (PNML) [30]. Both of these approaches are focused on providing either a specialised representation scheme, or a workflow management approach for components and/or services.

Our approach is more generic, and based on the provision of a standard pattern template library in UML and associated operators. Some of those operators may be used to support workflow, and PN models for patterns may also be constructed from their UML descriptions, as outlined in [7]. The PN models are useful to capture the semantics of the operators, and to undertake *what-if* investigations when combining operators. The availability of UML templates will make our approach more widely deployable, and may be used with a number of existing toolkits such as Rational Rose or TogetherJ (a survey can be found in [8]).

The utilisation of languages such as Java (such as the CoG [13] interface to Globus) and the central interest in Web Services [12] identifies the importance of using object-

oriented/component-oriented design approaches. Various tools are currently available which can take UML diagrams and generate code fragments for these technologies. We therefore feel that a representation centered on UML is easier to translate into working designs.

2.2.2 Dynamic Reconfiguration and Adaptability

Another strong contribution to enrich distributed environments came from *dynamic reconfiguration techniques*, their importance being due to the impossibility of predicting all possible configuration options at the time the system is first designed. The challenge is to allow new configuration options at run-time without disturbing the parts of the running application which are not involved in the reconfiguration process, and also keep the overall application's consistency. A common definition of reconfiguration is

Reconfiguration is to modify either the structure, or the topology, or the implementation of a (distributed) application.

These types of reconfiguration may happen at different levels of a distributed computing platform. For example, in hardware resource management, which tries to provide users with the best possible quality of service according to the available resources, it may be necessary to run some parts of the application in different machines ("modification of the topology"). On the other hand, at the end-user level, it may be necessary to introduce new logical resources or to change the existing resource associations ("modification of the structure"). Finally, existing middleware systems which support reflection may allow changing the code of some resources ("modification of the implementation").

Besides traditional *dynamic reconfiguration techniques* based on *scripting languages*, the *component paradigm* itself simplifies the dynamic reconfiguration process. For example, component platforms already support, at run-time, component replacement or system extension with new components (i.e. "modification of the structure").

An ultimate goal related to reconfigurability is to build *self-adaptable systems*. Besides having dynamic reconfiguration capabilities, those systems are automatically able to decide when to launch the reconfiguration process. Such is central to the support of the increasingly important area of Autonomic Computing [187].

Problems in Dynamic Reconfiguration

One problem in dynamic reconfiguration is to change the functionality of the environment by changing the implementation of the running code. *Reflection techniques* [188] intend to provide precisely this ability, which is to be used when "plug-in" capabilities are not sufficient. The ultimate goal is to allow users to apply reflection either over the "components" themselves (if they are seen as "grey boxes") or over the "glue code" that binds components together. Some works go a step further about what can be changed and consider the system under an architectural perspective². As such, the interactions between components are defined as "first class entities",

²In the discipline of *Software Architecture* [99], a system has two types of entities: components and connectors; components are executable entities each obeying some particular logic, and connectors contain the components' interactions – connectors bind the components together.

and reflection is used to change dynamically those entities³. In this way, interaction patterns are encapsulated in those “first class entities” (i.e. connectors) which can then be reused [189]. Furthermore, these patterns themselves may be dynamically “adapted” to, for example, best accommodate a new component (e.g. that replaced another one with which it is not totally compatible).

A second problem has to do with the overall effects of reconfiguration over the running application. One dimension of this problem has to do with the consistency of the application, i.e. that after reconfiguration the system still performs correctly (i.e. it satisfies the specifications). For some solutions this requires reaching a consistent state from which it is then possible to reconfigure the system [190]. For other solutions, the problem is solved through the explicit representation and management of dependencies in a graph.

Other dimension has to do with the *quality of service* (QoS) of the non-functional properties of the system, for example performance or reliability. In this case, a user may require that a specific level of QoS of the system has to be preserved. This may imply that the underlying system itself has to proceed with further reconfigurations than those required by the user, so that some QoS may be guaranteed. Otherwise, the system should inform users of the consequences and ask for alternatives. At a higher level, some works study the reconfiguration problem considering the semantics of the application (e.g. select a component which is semantically compatible with the overall environment or even adapt a new component so that the whole semantic behavior is preserved).

Finally, an additional problem is how to control the reconfiguration process itself: do it at once; do it in stages; which parts are automatic; which ones are controlled by a user, i.e. to allow the explicit coordination of the reconfiguration process; separate the reconfiguration policy from the reconfiguration process and give the possibility to dynamically change that policy.

In our work, we propose an approach towards (dynamic) reconfiguration. Specifically, we propose reconfiguration capabilities based on pattern manipulation through pattern operators. Namely, the unit of reconfiguration is a pattern (either a Pattern Template or a Pattern Instance), where its structure can be changed independently from its ruling behaviour, and vice-versa (examples are presented in section 5.3.2). Moreover, each pattern can be directly reconfigured, even if it is embedded in a Hierarchic Pattern, restricting in this way those changes to a sub-domain within the application (which in turn is represented by that Hierarchic Pattern). For instance, the example in section 7.3.6 illustrates the case of reconfiguring an embedded pattern with no consequences for the Hierarchic pattern that represents the overall application (namely, the number of remote users to a *Steering Interface* is modified with no implications on the configuration of the PSE application including that interface). Nevertheless, a thorough study on the associated problems of pattern-based reconfiguration still has to be addressed in our future work.

³The objective and the concepts behind this approach are quite similar to the “Control-driven Coordination Models”.

2.2.3 Coordination Paradigm

Another contribution for the development of distributed applications, came from the *Coordination Paradigm* [214]. This paradigm is concerned with high-level problems within distributed environments, like creation/destruction of coordinated entities, control of communication flow between these entities, or control of distributed execution and synchronization. Being such a general concept, coordination is also related to the specific problems of dynamic composition and reconfiguration.

The general discipline of *Coordination Theory* [215] studies

the body of principles about how activities can be coordinated, i.e. how actors can work together harmoniously (this includes conflict resolution and cooperation).

The theory has many practical applications in the distributed systems domain, like providing the necessary features to allow a set of human users to effectively cooperate on a task⁴, or identifying the adequate (programming) models for process cooperation in concurrent, parallel, and distributed systems. In particular, *Coordination Models and Languages* [214, 216] had the major role of highlighting the importance of clearly identifying the coordination issues, i.e. where an “actor” (e.g. a process) is not working alone anymore but is contributing to a wider coordination policy.

Contrarily to *Data-driven Coordination Models* (e.g. Linda model [109]), *Control-driven Coordination Models* [214] completely separate computation from coordination: the computational entities are considered as black-boxes with defined interfaces where internal data is irrelevant for coordination; the coordination rules (patterns) are encapsulated in a separate entity. Namely, and using the words of the *Component paradigm*, the entities are separated in “computational components” and “coordination components”. The *coordination language* (e.g. [212]) used to program the coordination components is completely independent from the language(s) used to build the computational components. The “composition of components” (or services) means to define, inside a coordination component, the *coordination patterns* that represent the “harmonious” work of a set of components (or services, e.g. [213]). These latter components may be computational components or even other coordination components.

Reuse is also present in those models, namely, reuse of computational components, and reuse of coordination patterns (because the same patterns of interaction occur in many different problems). Furthermore, those types of coordination languages are adequate to control the reconfiguration process itself [211].

Such concerns have motivated our interest on the availability of Behavioural/Coordination Patterns for application development.

2.3 Solutions for Structure and Interaction Reusability

The three approaches described in section 2.2 define major contributions for the development and control of complex applications. The Component Paradigm supports the configuration of a system into components and connectors, where the latter ones represent the interactions

⁴Problem studied by the CSCW systems (*Computer Supported Cooperative Work*).

between the former. Components/services represent a good abstraction for reusability, which has resulted on the existence of many component-oriented systems, and recently on service-oriented systems.

The Coordination Paradigm, in turn, aims to provide more adequate abstractions to represent and manage the interactions between the elements of a system. The Component and Service Paradigms and the Coordination Paradigm are easily intertwined because they both promote a separation between computational units and their interactions. In fact, several coordination models and languages were specifically developed for component systems, providing a system which provides reusability of computational units as well as of interaction “idioms” [212,213].

The Reconfiguration Paradigm and the Coordination Paradigms are related as well [211]. For instance, dynamic reconfiguration requires a specific form of coordination, where the evolution of the system has to be controlled in a way that the system consistency is guaranteed. The reusability of dynamic reconfiguration protocols is a major asset for a dynamic distributed system such as the Grid.

In the context of our model, we propose an initial approach towards dynamic reconfiguration based on Hierarchical Patterns, each one reconfigurable through Pattern Operators (see 5.2 and 5.3 in particular). Namely, our approach supports the combination of Behavioural Patterns into hierarchies, where each individual pattern is still directly manipulable through operators within the hierarchy. However, the (hierarchical) composition of diverse Coordination/Behavioural Patterns raises several complex interdependencies problems by itself, which could not be addressed in the context of this thesis. Such problems, and the manipulation of patterns towards dynamic reconfiguration, will be the subject of future research in the context of our proposed model.

To summarise, reusability appears as an important feature of the systems that rely on the three mentioned paradigms. It is not acceptable anymore to build complex systems from scratch, due to time and cost restrictions. Not only the computation units have to be reused, but also the configuration of an application and the interaction rules between its elements, should be reused as well. Specifically, due to the complexity of distributed systems and the Grid, it is desirable to provide mechanisms for reusing and combining common coordination schemes (e.g. defining flow dependencies between components/services).

In this section, we present two other abstractions that have been used in parallel and distributed systems. The abstractions are:

Skeletons mainly used on parallel programming and represent abstractions at the programming level;

Patterns which started to be applied at the design level of centralized and distributed systems, but which nowadays, are becoming first class entities throughout the development and execution life cycle of distributed applications.

One of the major contributions of our work is precisely the manipulation of patterns as first class entities, namely in Grid-supported PSEs, from the design phase, to the execution phase. Our proposed model also contemplates pattern manipulation supporting dynamic reconfiguration.

It is worthwhile to mention that, in some systems, the distinction between skeletons and patterns is not clear. Early versions of those systems were directed specifically to parallel applications, so they were based on the skeleton abstraction. Recent versions aim at giving support to parallel and distributed environments like the Grid, and sometimes the “pattern” word has simply replaced the “skeleton” word. Nevertheless, patterns and skeletons have, in some way, many similarities. For example, the *Pipes-and-Filters* [10] and the *Master-Slave* pattern (see section 3.2.4) correspond to the *pipeline* and *farm* skeletons respectively (these are described ahead).

However, in [47], the editors make a clear distinction between skeletons and patterns:

- a) the description of skeletons is formal, whereas pattern descriptions are loosely described either in English and/or a combination of UML [68] diagrams;
- b) a design pattern has consequences across several phases of the development cycle, whereas skeletons are used as a programming abstraction;
- c) skeletons are directed to the design of high-performance systems, whereas patterns are more general since they may represent common general requirements in distributed systems (e.g. fault-tolerance, timeliness, and quality of service).

2.3.1 Skeletons

The work on skeletons originates in the parallel computing community, and is based on the use of *algorithmic skeletons*⁵. The predominant motivation behind this has been the need to overcome the difficulty of constructing parallel programs – by capturing common algorithmic forms which may subsequently be used as components for building parallel programs [167], [24].

Such skeletons are expected to provide parameterisable abstractions that may be composed – generally using a functional programming language. A skeleton is expected to be transparent to an application user (and may come with a pre-packaged implementation). Skeletons are viewed formally as polymorphic, higher-order functions – which may be repeatedly applied to achieve various transformations (on data structures such as lists). In fact, in skeleton-oriented functional languages, the functional programmer considers a skeleton to be simply a polymorphic higher-order function which can be applied with many different types and parameters. As such, programming with skeletons, as with high-order functions, is “to define each concept once and to reuse it many times”.

Being based on high-order functions, many skeleton systems use functional languages as the host language [155,157,159]. However, to increase efficiency, some systems chose to extend imperative languages like C and C++ [152, 160, 161, 164]. Nevertheless, those systems offer typically a closed collection of skeletons which the application programmer can use, but the addition of new skeletons usually implies a considerable effort.

In general, the main points behind skeletons are [151]:

⁵More generally, skeletons may be classified as [179]: *algorithmic skeletons*, which encapsulate control structures that represent some standard algorithm (complete or a fragment); or *homomorphic skeletons*, which take into account geometric information, being associated to particular data types (lists, arrays, etc.).

1. A complex parallel application can be coded at high-level by instantiating and composing available skeletons.
2. A cost model may be associated to a skeleton which allows the programmer to make sensible decisions during the software development process.
3. In some situations, cost models may be used to manage resource optimisation automatically.

Additionally, and related to the first point defined above, skeleton systems may be divided into [151]:

Flat Skeleton systems The structuring of a parallel application in these systems (e.g. [152, 155, 167]) is based on a single skeleton, which limits expressivity. Although in some of those systems is possible to add new skeletons, such hinders their simplicity.

Systems providing Skeleton Nesting These systems support an adequate composition of skeletons to form more complex applications [151, 157]. Due to the similarities of this kind of skeletons systems to our hierarchical pattern-based model, the examples discussed in section 2.4 reference systems that support such skeleton nesting.

Specifically, skeleton nesting is supported through *skeletal composition languages*. *Basic skeletons* representing simple recurrent parallel patterns are the building blocks of those languages. An application's parallel structure is expressed only as a composition of basic skeletons. In skeletal composition languages, a cost prediction of the whole program is still available at the programmer level, and it can be derived from the cost of the program's constituent skeletons. Optimisation tuning is hidden by the implementation. However, skeletal composition languages require an adequate choice of the basic skeleton set, a reliable cost model, and an efficient implementation. Examples of basic skeletons are [151]:

Data parallel skeletons Represent the application of general operations over large data structures whose sub-structures are processed in parallel. For example:

- *map* – a particular function is applied to each element in a data structure (e.g. application of a function to a list producing a list of the same size);
- *filter* – all elements that satisfy a specific predicate are filtered;
- *reduce* – this skeleton corresponds to a reduction, namely, an associative operator is applied to a data structure generating a single value as a result.

Task parallel skeletons Represent the parallel execution of a task by dividing it into sub-tasks. For example:

- *divide_and_conquer* – a split function divides a problem into a set of sub-problems that upon being processed in parallel, their produced results are combined by a join function into a new (sub-)solution; such processing is recursively applied to all sub-problems.
- *farm* – for each independent data item, the controller of the task skeleton selects one worker (from a pool of workers) to execute that data; the workers execute in parallel, and all their produced results are afterwards gathered together.

2.3.2 Patterns

A Pattern [9, 10] encodes a commonly recurring theme in service or component composition. It allows good practice to be identified, and shared across application domains. A pattern is generally defined in an application independent manner, and used to encode particular useful behaviours. Patterns are particularly useful for configuring and specifying systems that are composed of independent sub-domains. Patterns are aimed at capturing some generic attributes of a system – which may be further refined (eventually) to lead to an implementation. These are important requirements for Grid computing applications, which generally need to operate in dynamic environments, as proposed in the present work.

Libraries of common “patterns” for designing software allow the developers to select more adequate patterns. Design patterns’ documentation is in most cases rather informal, namely in textual form, and/or is presented using UML diagrams. Additionally, one way to select the adequate patterns to define an application is to use a pattern language. For example, in [185] a pattern language is defined for parallel application programs. Specifically,

“A pattern language is a collection of design patterns that are carefully organised to embody a design methodology. A designer is led through the pattern language, at each step choosing an appropriate pattern, until the final design is obtained in terms of a web of patterns” [185].

Although it was not our goal in this work to provide a pattern language for Grid environments, the set of selected patterns (discussed in section 3.2) provides, in our opinion, adequate expressiveness for the configuration of different kinds of typical applications in the above environments.

Additionally, Patterns applicability ranges from high-level strategies for organising software to low-level implementation mechanisms. In the latter case, patterns are called “idioms” and represent language-dependent techniques to model objects. Idioms are being applied in a variety of contexts, from concurrent programming in Java to distributed programming in CORBA. Patterns started to be identified and applied in object-oriented user interfaces. In this case, patterns’ main quality criteria were usability, extensibility and portability. Soon, patterns’ suitability to parallel and distributed systems was also recognised, and patterns for diverse domains were defined. This is illustrated in the following sub-sections.

Parallel patterns

Besides skeletons, the pattern concept started also to be used by some parallel research groups to capture concurrency and parallelism characteristics (e.g. [175, 185]). However, initially, the main goal of pattern usage in the parallel computing domain was targeted to issues of synchronisation and non-determinism which are more relevant to distributed computing. Recent research works on patterns in the parallel community, on the other hand, have been tackling concerns of High Performance Computing, namely the specific facets of concurrency and parallelism [118, 143, 148]. Consequently, these research directions show pattern usage where the connection to skeletons has become increasingly apparent.

Examples of parallel patterns are [110, 118, 144]: mesh design pattern; pipeline; master-slave; work-queue; divide-and-conquer; wavefront; fork/join model; workpiles and meshes. Additional parallel patterns are discussed also in [186].

Patterns for Object-Oriented Middleware

Patterns for Object-Oriented (OO) middleware represent recurring structure and interaction schemas in common OO middleware like CORBA, Web Servers and Peer-to-Peer systems.

Several examples can be found in [110], such as:

- *Service Access and Configuration Patterns*:
 - *Wrapper Facade design pattern* – encapsulates the functions and data provided by existing non-object-oriented APIs within more concise, robust, portable, maintainable, and cohesive object-oriented class interfaces.
 - *Component Configurator design pattern* – allows an application to link and unlink its component implementations at run-time without having to modify, recompile, or statically relink the application. Component Configurator further supports the re-configuration of components into different application processes without having to shut down and re-start running processes.
- *Event Handling Patterns*:
 - *Reactor architectural pattern* – allows event-driven applications to demultiplex and dispatch service requests that are delivered to an application from one or more clients.

Workflow Patterns

Workflow Patterns identify common requirements and control flow schemas in state-of-the-art Workflow Systems. The work developed by Wil van der Aalst on workflow patterns [79] presents an exhaustive study of common characteristics (e.g. interactions) in workflows, for instance, in the dimensions of

Control flow These define common dependencies between workflow tasks concerning control flow.

Data representation and dependencies These characterise the way data is commonly represented and utilised in workflows, namely: the way data elements are perceived by the tasks in workflow (i.e. data visibility and data interaction); the way data is transferred between workflow tasks; and finally, the way data elements may have influence upon workflow execution (e.g. over control flow between tasks).

Necessary resources These capture common ways of resource representation and usage in workflows, e.g. the necessary allocation of resources, tasks, etc., as well how delegation is supported.

For instance, examples of control flow patterns discussed in [79] are:

- *Basic Control Flow Patterns* – these are supported, for example, in the Triana workflow system [82,201] and in the Karajan workflow system [207]. Examples:
 - *Sequence* – the tasks in a workflow are processed sequentially, namely, after the completion of a task, the execution of the next task in the sequence is enabled.
 - *Parallel Split* – upon the execution of a thread of control in the workflow process, multiple threads of control are generated allowing the parallel execution of multiple tasks in the workflow.
- *Advanced Branching and Synchronization Patterns*:
 - *Multi-choice* – defines the possibility of selecting one or more branches, among several branches in the workflow process. For example, in a point in the workflow process the evaluation of workflow control data defines which branches in the workflow process to select resulting on the activation of the correspondent workflow tasks. This pattern is fully supported in Karajan [207] and Triana [82] supports a limited version (through a “if-then-else” Control Unit).
 - *Synchronizing Merge* – represent the convergence of multiple paths in the workflow process to a single point supported by a single thread. The pattern defines that on the existence of multiple threads resulting from the process of (some of) those multiple paths, those threads have to be synchronised before processing the next (single) thread in the workflow process. It is assumption in the pattern that an activated branch cannot be re-activated while the synchronisation (i.e. merge) of all branches does not take place. The concept underlying this pattern is used in the implementation of our work over Triana (see section 6.4), namely for the *Repeat* Execution Operator where it is necessary to guarantee that all tasks within a pattern (or all embedded patterns within a Hierarchic Pattern) have terminated before the next iteration in the *Repeat* operator is activated (which will generate another execution of those patterns).

Design Patterns for Computational Grids

The importance of Patterns for Grid computing was first extensively discussed in [47]. Namely, in [34] the authors identify a set of *Service Design Patterns for Computational Grids*. These patterns identify how applications may be composed, shared, and managed over a Computational Grid. Examples of patterns in [34] are:

- *Broker Service Pattern*: provides a service to support a user application to discover suitable computational services. A Broker may utilise a number of other services to achieve this objective.
- *Service Adapter Pattern*: attaches additional properties of behaviours to an existing application to enable it to be invoked as a service. This pattern is present in some examples in this thesis.

2.4 Skeleton/Pattern-based Models and Systems

At the time of the central definition of our work [37, 44, 45] very few systems provided design patterns as building abstractions for application development. Specifically, to the best of our knowledge, none of those systems provided patterns for Grid application development in the way we propose in this dissertation (e.g. patterns as first class entities for the whole life cycle of application development). However, the inclusion of skeletons already existed for some parallel systems which have been extended to the Grid domain. Nevertheless, skeletons in those systems are not manipulable entities through operators, e.g. for execution control and reconfiguration, as we propose in our work.

In this section, we describe shortly some of the existent systems at that time, whose authors highlight the importance of providing skeletons/patterns for reusing common interaction schemes in distributed and parallel applications. Some works were selected based on the existence of similarities to the model described in this thesis towards highlighting the relevance of our work. For example, the defined skeletons in the *P3L* parallel programming language can be mapped to our *Structural* and *Behavioural Patterns* as will be discussed in the example in section 7.4.

2.4.1 Skeleton-based Models and Systems

This section describes a few models and systems that use skeletons as the high-level programming abstraction.

Pisa Parallel Programming Language (P3L)

P3L [150, 151, 153] is a skeletal composition language where programs are composed of set of code fragments and a skeleton description that describes how the fragments are composed into a complete program. The language provides nesting of pre-defined basic data and parallel skeletons, allowing complex global parallel structures of a program.

P3L is suited for mixed task and data parallelism applications since applications are configured according to a two tier structure. Namely, task parallelism is exploited at a coarse level among groups of processes, and these exploit data parallelism (*data parallel tasks*). P3L is also intended for applications that have a static and predictable parallel structure and work on a stream of independent input data sets. Furthermore, P3L supports performance tuning at the user level through cost models associated to the basic skeletons.

Structure and Behaviour

P3L presents a clearly defined model where the parallel structure results from the composition and nesting of *task (TPSs)* and *data parallel skeletons (DPSs)*. DPSs abstract array partition and alignment of dense multi-dimensional array structures. TPSs define the overall parallel structure connecting DPSs. In the model, behaviour is hidden in TPSs, DPSs, and in *control parallel skeletons (CPSs)* which can be freely nested because they do not change the parallel structure of the application.

- DPSs' behaviour is to parallelize a function which is applied to the different parts of

the dense multidimensional arrays. Examples: *map* distribute input data according to a user specified pattern creating a tuple of aligned arrays; *reduce* “sums” the elements of an array using a binary operator; *scan* computes the parallel prefix of an array using a binary operator; *comp* models usual functional composition. DPSs can be nested in DPSs and in TPSs.

- TPSs exploit parallelism between the execution of *data parallel tasks* (instances of DPSs) on a stream of homogeneous independent input data. TPSs generate a stream of results. Examples: in the *pipeline* TPS, a sequence of skeletons (DPSs or TPSs) execute concurrently defining independent stages of a computation; the *farm* TPS replicates a skeleton (a DPS or TPS) in a pool of identical copies (the *workers*). Different workers compute independent data items of the input stream. The workers are scheduled in order to guarantee load balancing, and their outputs are merged forming the farm output stream. TPSs can be nested in TPSs but cannot be nested inside DPSs.
- The CPSs are: *seq* wraps sequential code which will be used to instantiate truly parallel skeletons; *loop* iterates the execution of a skeleton on the received input until a condition is verified. In this case, a single input can cause several executions of the skeleton controlled by the loop. CPSs can be freely nested in other CPSs, DPSs, and TPSs.

The programmer defines the nesting of skeletons by invoking one or more skeletons inside a skeleton declaration, and the pre-defined semantics have to be guaranteed (e.g. TPSs cannot be nested in DPSs, as cited above).

In terms of execution, skeletons can only work on independent input data sets, and a skeleton’s arguments of a skeleton instance can only match the pre-defined types in P3L. Moreover, each P3L computation can only have exactly one source and one sink data parallel task (i.e. an instance of a DPS). TPSs, in particular, are supposed to produce as many output values as the number of input values which are consumed. Communication, in both TPSs and DPSs, is hidden from the programmer. Namely, the actual way each input is fetched in and results are passed on to next DPS in the structure, according to what was defined by the programmer, is implementation dependent.

Similarly to our model, P3L also provides configuration constructors supporting hierarchies, namely TPSs and DPSs. However, DPSs are not general, as they are meant for dense multidimensional arrays. Nevertheless, DPSs are abstract in the sense that they hide parallelism – the actual mapping of a DPS into a disjoint set of processors is implementation dependent. However, there are no explicit operators to manipulate skeletons contrasting with the model presented in this thesis where operators can manipulate patterns.

Moreover, TPSs do represent the pipeline and farm general parallel patterns, and execution is based on the *Streaming (data-flow) behavioural pattern*. However, in the farm TPS, the selection of which worker to run next is the responsibility of the controller of the farm whose actions are implementation dependent.

Program reconfiguration in P3L, in turn, requires recompilation of the code. Nevertheless, the change of the global structure of the program may be done with few coding if new complex global parallel structures of a program may then be defined.

In principle, the P3L language is not easily extensible because a program relies on a particular abstract machine. Specifically, a P3L program is mapped onto the underlying hardware architecture by creating an abstract machine tailored for the skeleton and execution environment. New skeletons would imply a skeleton designer to create new abstract machines. Nevertheless, the existence of a fixed set of available skeletons is a feature, not a weak point, since the authors claim that this is the way to guarantee the best performance.

Finally, the authors also claim that the techniques used are scalable to WAN scale meta-computing systems, but such developments were still not available at the time of this writing.

Implementation

Anacleto is a cross-compiler of the P3L programs in a SPMD program written in C + MPI (standard message-passing library MPI [165]). Implementation is specific to an architecture of Linux Clusters with MPI. Translation of P3L programs is accomplished using a library of *implementation templates* (the *template library*) which consist of a generic implementation of a skeleton. The templates can be parameterised, for example, with fragments of sequential code, for the definition of the input/output types, to specify the number of works, etc. The library has several templates for the same skeleton, each one providing different implementation strategies and an associated cost model. *Anacleto* was implemented in a modular way to allow new skeletons and templates to be inserted.

Skil and a Skeletal Parallel Programming Library

This section describes some of Kuchen’s work on skeletons, from the language *Skil* to a system which provides a library of skeletons.

Skil (*Skeleton Imperative Language*) [160] is a language that provides algorithmic skeletons to the programmer by supporting: functional features like high-order functions (parameters may be other functions or even partial applications), a polymorphic type system, and the definition of *distributed (parallel) data structures*. The selected approach was to provide *Skil* as an imperative language based on a subset of the language C, overcoming the inefficiencies of pure high-order functional languages. The low-level support was based on MPI.

Although the language *Skil* proved to be an efficient way to support parallel programming, according to the (co-)author of [163, 164], the availability of a library of skeletons implemented in C++ has the advantage of attracting more typical parallel programmers to the skeleton’s inherent benefits. Namely, C++ is a popular language among the parallel programming community, and providing skeletons as C++ templates reduces the effort to learn “skeleton-based programming”. In this way, programmers can manipulate high-level abstractions to define parallelism without the burden of low-level implementation details, and without significant performance loss.

Structure and Behaviour

Similarly to the aforementioned P3L language, the skeleton library described in [161, 162, 164] is based on a two-layer model, consisting of *Task Parallel Skeletons (TPSs)* and *Data Parallel Skeletons (DPSs)*.

In general, the main concepts result from: the integration of data parallelism from the *Skil* language; well-known task parallel skeletons such as pipeline and farm; and the two-tier model of P3L. The parallel structure results from task parallelism on the outer level and an atomic task

parallel computation may use data parallelism inside. This means that, like in P3L, nesting is provided by both invoking TPSs and DPSs inside TPSs, but also by enclosing skeletons in C++ control structures like loop and conditionals. Similarly to P3L, TPSs cannot be nested in DPSs. According to the authors [161] there are many algorithms in which data parallel components exist within a task parallel (e.g. pipeline) framework, but there are no realistic examples where the reverse holds.

The skeleton library also provides benefits regarding flexibility. Specifically, skeletons' argument functions are not restricted to C++ functions, but can be partial applications as well. Since the details of a skeleton's underlying structure are dependent on the skeletons' arguments, some of those details are then dependent on parameters of the partial applications which are computed at runtime.

Although the behaviour is encapsulated in TPSs and DPSs and in the enclosing C++ code, the stream processing model underpins the task parallel components. Contrary to P3L, there is no restriction on the number of output data items produced as a result of the input data items consumed and, at the time of this writing, the version of the library does not provide a skeleton-based cost analyser and a corresponding optimiser.

Skil provided two distinct classes of data parallel skeletons, namely *computation skeletons* and *communication skeletons*. Computation skeletons process the elements of a distributed data structure in parallel. Communication consists of the exchange of the partitions of a distributed data structure between all processors participating in the data parallel computation. There is no implicit communication like accessing elements of a remote partition. Starvation and deadlocks are avoided because partitions are exchanged in a synchronised way and there are no individual messages. Examples of communication skeletons: "permutePartition", and skeletons that represent MPI collective operations like "broadcastRow" and "gather".

However, in the library [162, 163], communication skeletons are available as operations over the DPSs. The available DP skeletons are: *DistributedArray* (DA) and *DistributedMatrix* (DM). There are many operations which allow, for example, the access to attributes of the local partitions of a distributed data structure. The DA skeleton is represented as a C++ template which provides methods (to implement the operations) like *getSize* which returns the number of elements of the DA, *map* that applies a function to all elements of the DA, or *fold* which combines all the elements of the DA by a binary function passed as argument to the method.

Task Parallel Skeletons (TPS) are also implemented as templates which can be parameterised.

- The simplest TPS is the *Atomic* skeleton which takes a sequence of inputs and transforms it into a sequence of output values by applying a unary function to each of the inputs. The atomic skeleton may have data parallelism internally.
- The *Filter* skeleton is more general allowing, for each input, an arbitrary number of output values to be produced (including 0). Inputs are consumed using the auxiliary operation *get*, and outputs are produced using operation *put*.
- The *Pipe* and *Farm* skeletons are similar to the ones in P3L.
- The *Par* skeleton (parallel composition) is similar to the Farm skeleton, but the sequence

of inputs is forwarded to all the workers. The outputs produced by the workers are merged non-deterministically and propagated.

- The *Loop* TPS encapsulate in the body another TPS where output values produced by the body are propagated to the next skeleton and/or given back to the body, depending on two boolean argument functions.
- The *Search* is also a farm-like skeleton which solves a sequence of search problems. The problem is divided into sub-problems which are given to the workers. A different data structure is used for each problem (e.g. a stack leads to k-parallel depth first search, a queue leads to k-parallel breadth first search, and a heap leads to (a variant of) best first search).
- The *Branch and Bound* TPS is similar to the Search skeleton but where a boolean function *less* is used to identify the most promising sub-problems which will then be given first to the workers.

Implementation

According to the authors [164], the implementation of algorithmic skeletons is based on three features: parametric polymorphism, higher-order functions and partial applications. C++ templates were crucial to support all those features. High-order functions and partial applications were possible as a result of C++ operator overloading (in particular overloading of the parenthesis operator - *operator()*).

The skeleton library was implemented on top of MPI with the consequent advantages of platform independence and reduced performance penalties. There are no individual messages – all the messages are transparently coordinated within skeletons' boundaries. Such coordination requires, in some cases, a rather sophisticated communication protocol (that is the case of the controller, an auxiliary process which is responsible for coordinating the workers of a task parallel skeleton). Such complex coordination is necessary because, contrary to P3L, processes in a TPS are not restricted to produce the same number of output values as the number of input values that were consumed.

eSkel

The *Edinburgh Skeleton Library* (*eSkel*) [169,170] is a C library of algorithmic skeletons built upon MPI. Similarly to other systems, skeletons abstract common recurring patterns of parallel behaviour which can be parameterised with application specific functions. Like in Skil, skeletons are presented in the form of a library in order to avoid the introduction of any new syntax. Examples of implemented skeletons are pipeline, task farming and butterfly style divide-and-conquer.

The underlying conceptual model of eSkel is that of SPMD distributed memory parallelism, quite common in MPI applications. Although skeleton implementation is transparent to the user, MPI's collective operations are available (e.g. MPI_Broadcast, MPI_Reduce) meaning that the user code may invoke them directly. Recent versions of eSkel [171,172] support skeletons over the Grid to enhance application's performance. The approach considers single skeletons which span the Grid, and modelling techniques are used to estimate performance.

Structure in eSkel is defined by skeletons which represent a group of processes each one calling the collective operation that represents the skeleton. According to the skeleton's semantics, during the call of the collective operation the members of the group are grouped and regrouped by the implementation. For example, each sub-group may execute a stage in a pipeline; the authors call "activity" to a sub-group of a skeleton. The model also supports skeleton embedding, where in the previous example, a stage may be a skeleton itself.

Each group representing a skeleton has a special associated process called the "communicator" which is passed as argument to the collective call. By accessing this communicator, the members of the group may communicate explicitly with other members, with the guarantee that communication is restricted to the group boundaries. In fact, eSkel makes a clear distinction between explicit and implicit communication. Implicit communication represents the necessary communication to support the skeletons semantics, i.e. the interactions between activities (e.g. communication between two consecutive stages of a pipeline). Explicit communication, as mentioned above, circumvents the skeleton's default behaviour, allowing a process in a activity to communicate with another process in the same activity.

Behaviour is encapsulated in a skeleton and is the result of implicit and explicit communication. As said before, each stage may be represented as a skeleton itself, or may show internal parallelism through direct calls to MPI. Low level details such as task distribution, load balancing and collating and storing results, are transparent to the user. Similarly to Skil, eSkel allows that activities may produce more than one result or none result at all.

In general, pre-defined semantics are related to skeletons themselves, i.e. they define the way in which a skeleton's activities may interact. Spatial constraints determine the activities with which each activity may interact and the directions these interactions may take. These are called "partner activities" (e.g. two consecutive stages in a pipeline). Temporal constraints, in turn, determine the allowable orderings of interactions between partners. For example, in a particularly strict form of pipeline, it may be required that a stage interacts first with its predecessor then its successor, in strict alternation. Our work may support this kind of behaviour through Behavioural Patterns and through implementation dependent "Task triggers"(through "trigger nodes" as defined in Chapter 6).

Reconfiguration is one of the goals in the under development version of eSkel. Skeletons are modeled in a generic way to obtain significant performance results which may be used to reschedule the application dynamically. At the time of this writing, it was still not possible to reconfigure a skeleton dynamically forming a new one.

Implementation

Each activity is associated with a new communicator, allowing to trace a stack of communicators for the whole embedding. Our approach also relies on a *pattern controller* for each *Pattern Instance* (i.e. *Structural* plus *Behavioural Pattern* combination). In a *Hierarchical Pattern Instance* the result is a tree of connected pattern controllers (e.g. supporting execution control as presented in Chapter 6).

Parallelisation in eSkel is supported by multithreading over MPI, and no new abstract data types were introduced for distributed-shared data. Skeletons in eSkel add the facility to move data between activities, following the skeleton specification, without the need to explicitly in-

voke MPI communications. As such, it is possible to reuse existing components.

Other Systems

The *PAS* system [144] is based on algorithmic skeletons [167] providing a number of communication interfaces tailored for specific parallel structures such as workpiles and meshes. Applications are written either using a specification language that includes special constructs for skeleton information or using template C++ code and instantiating the correct communication interface. It is unclear if the specification language in *PAS* can be easily extended to incorporate new communication interfaces. Nevertheless, the subsequent system, namely *SuperPAS* [149] overcome such limitation.

The *ASSIST* [173, 174] system is a software development system based upon integrated skeleton technology supporting effective reuse of parallel software across different platforms, in particular large-scale platforms (and recently grids). Application design in *ASSIST* consists of defining generic graphs of parallel components. To this extent, and besides supporting semantic definition of several skeletons as particular cases, the *ASSIST* system defines a new paradigm named “parallel module” (*parmod*) to express more general parallel and distributed program structures (including both data-flow and non-deterministic reactive computations). Additionally, external objects (e.g. shared data structures and CORBA abstract objects) can be used within *ASSIST* modules, and *ASSIST* applications can be reused and exported as components for other applications.

2.4.2 Pattern-based Models and Systems

The goal of this section is to describe examples of tools that provide patterns for application programming. Namely, we mention a simple design tool (*Model Maker*) without support for distributed programming, and a tool for parallel programming (*CO2P3S*).

Model Maker

The *Model Maker* tool from Borland [182] includes design patterns as an integral part of its modelling engine supporting the Delphi language. Patterns are defined at the same level as classes and units, and they may interact with or extend other patterns. In this way, the user benefits from expert knowledge by reusing successful designs and architectures, making the system itself reusable.

Design patterns also improve the documentation and maintenance of the built system since they provide an explicit specification of class and object interactions and describe their main purpose. The application of a pattern requires the specification of its context by insertion of user code into the model. Specifically, it may be necessary to include classes, member or code sections in methods. The pattern helps the user on deciding what to insert where. Furthermore, the patterns are “active” (named “active patterns”) – they “stay alive” by detecting changes in the code associated with the pattern.

The user may create new patterns through “Parametric Code Templates”, although these are not “active patterns”. Examples of the available patterns are mainly some of the ones de-

scribed in [9]: Adaptor, Mediator, Singleton, Decorator, Visitor, and Observer. Also available are the “Lock Pattern” that provides a mechanism to temporarily lock some aspect of a class (see description in [183]), and “Reference counting pattern” that “provides a mechanism to control the life-span of an object with reference counting” [182].

Enterprise and CO2P3S

The work developed in the project *CO2P3S (Correct Object-Oriented Pattern-based Parallel Programming Systems)* [115] aims at combining different abstractions to reduce the complexity of developing correct parallel applications. The abstractions are from the domains of object-oriented programming, design patterns, frameworks, and programming layers. The user is provided with a set of pattern templates, each one including several parameters that the user instantiates according to the domain-specific requirements of the application. Application examples can be found in [117].

The CO2P3S system was the successor of the Enterprise system [122, 123]. Although the Enterprise system offered a good tool support and users were able to quickly create a working parallel program based on patterns, it was not possible to tune its performance. Moreover, performance errors were easily introduced by the users since the Enterprise system required a subtle change to programming language semantics. The CO2P3S system overcome these problems, and also introduced the concept of *generative design pattern* [125]. This pattern generates code as opposed to simple descriptive design patterns, i.e. a custom framework is generated from the instantiated pattern templates. The framework has the advantage of encapsulating all of the parallel structure, including synchronization and communication code, with the guarantee of code correctness. The generated code serves as the highest of three layers of application code, where the lower layers are used only for performance tuning according to the application’s performance requirements.

Whereas CO2P3S provides only a pre-defined set of pattern templates, *meta-CO2P3S* allows a pattern designer to rapidly and easily edit existing patterns and create new ones. Namely, in [116] the success of using CO2P3S and meta-CO2P3S is demonstrated to generate structurally correct parallel programs from parallel design patterns, and the creation of a new design pattern named the Wavefront pattern is also described. This pattern captures the common behaviour of wavefront computations⁶ used in several parallel programs (e.g. “the Biological Sequence Alignment Using Dynamic Programming”, “the Skyline Matrix Problem”, and “the Matrix Product Chain Problem”).

CO2P3S/*meta-CO2P3S* concurrency results from using different processors to compute either multiple elements or groups of elements (forming a block) at the same time. In this case, the evaluation of elements in the boundaries requires values from adjacent blocks. Such boundary exchange defines the communication and synchronization structure. The authors demonstrate that the usage of Wavefront pattern results on code with good speed-ups on shared-memory computers. The pattern is applied in the resolution of three problems, namely sequence alignment, skyline matrices and matrix product chain.

⁶In wavefront computations, “each element computes a value that depends on the computation of a set of previous elements, and the computation typically flows from one region to another”. [116]

The basis of the CO2P3S parallel programming system, the *Parallel Design Patterns (PDP)* process is described in [118,124]. The PDP process is independent of programming language and parallel architecture. In the mentioned paper, CO2P3S is one implementation of this process, which generates multithreaded Java framework code for shared memory multiprocessor systems. A distributed shared memory implementation is also described in [121].

The PDP process provides a layered development model by combining different abstraction techniques commonly used to reduce the complexity of sequential programming, namely *object oriented programming*, *design patterns*, and *frameworks*. At the topmost development layer, the *Patterns Layer*, the user is provided with a set of *design pattern templates* – parametrized constructs based on design patterns – whose parameters may be instantiated according to the desired parallel structure of a program.

After selection of the pattern template and refinement of the pattern structure through parameter instantiation, the parameters guide the code generator. The result is a correct customized framework consisting of abstract classes, which encapsulates all of the structural details of the pattern, including communication and synchronization. Since the framework is specific to the selected parameters, better performance levels are achieved. Moreover, the generation of a correct framework saves the user from writing and debugging this code, simplifying the creation of a parallel application.

The application specific sequential code, defined by the user after parameter instantiation, is kept separated from the generated code. The specific code is provided in the framework as hook methods which are invoked by the parallel structure code at the appropriate time. As such, frameworks and design pattern are combined considering the application domain of the framework to be the implementation of a pattern.

Other Systems

The *DPnDP* system [143] uses a design pattern information to generate code for the pattern where all pattern-specific communication is handled automatically. Although the system allows the addition of new patterns, this is done only at the C++ framework level instead of providing tools that support extensibility. Furthermore, only the structure of new patterns can be added. Behavioural aspects, such as pattern-specific communication, cannot be added.

The *ObjectAssembler* [55] is an example of a visual development environment that provides a catalogue of patterns for connecting JavaBean components.

The goal of the *Enhance Project* [111] is to enhance the performance predictability of Grid applications with Patterns and Process Algebras.

The *Pacosuite* [54] tool supports component composition through *composition patterns* which define component interactions.

2.5 Summary

This chapter highlighted the importance of Software Engineering abstractions such as components and services, dynamic reconfiguration, and coordination models, on application programming. Moreover, the chapter also highlights the importance of higher-level abstractions

such as skeletons and patterns to reuse common interactions in distributed and parallel systems in general, and presents some programming systems providing those abstractions.

To conclude, and in order to further argue in favor of the relevance on the availability of skeletons and patterns for application development, we mention a study which evaluated the practical usefulness of those abstractions in the particular domain of parallel processing. Specifically, in [177] a detailed study is described to compare the relevance of patterns on the usability of two *Parallel Programming Systems (PPSs)*. The evaluated systems were the *Enterprise* system [122] which provided patterns for application programming, and a *PVM* [178]-like library of message-passing routines.

For that evaluation, students were divided into two groups, each one running controlled experiments over one of the two parallel systems. Three comparison categories were used that affect the assessment of PPSs, namely *performance*, *applicability*, and *usability*. The authors claim that *usability*, in particular, had not been commonly used for evaluation of parallel systems, contrasting with performance. In their opinion usability will have an increasing importance on the acceptance of a parallel system. The used assessment metrics for *usability* were [177]:

“learning curve, probability of programming errors, functionality integration with other systems, deterministic performance compatibility with existing software, suitability for large-scale software engineering, power in the hands of an expert, ability to do incremental tuning”.

This early study revealed that the usage of *Enterprise* provided more productivity gains even though performance was poorer. One of the contributions to that gain was that the system prevented several common parallel programming errors concerned with correctness. This was due to the fact that the implementation of patterns (tuned for parallel systems) was reused, and less errors were generated as a result. Another contribution was the availability of an integrated set of support tools supporting the application development process.

Moreover, in [121] it is also claimed that for many parallel applications it is more important to provide a rapid and correct development of a parallel application than to focus only on performance. Nevertheless, the tools have to support the tuning of the available patterns, the easy definition of new patterns, and the provided patterns have to support that the application parallelism is abstracted from the target parallel architecture so that it is possible to generate code to a different architecture.

Such concerns are extensible for patterns in Grid domains, namely in terms of their representativity for common interactions in Grid applications, the independence of patterns from the supporting Grid platform (e.g. in terms of the mapping to resource managers), the possibility of adding new patterns, and the tuning of patterns (e.g. concerning Quality of Service issues).

Therefore, we argue that in Grid environments, e.g. whose users are aware of the required structures/topologies to configure their applications as well as of the required coordination schemes for their behavioural dependencies as proposed in this work, patterns are an important contribution for Grid programming.

At the time of the first definition of our work [44] the use of skeletons and patterns on Grid development environments was still limited. Namely, tools available for skeleton languages did not connect to Grid middleware, such as Globus or UNICORE, although skeletons

based approaches do provide a useful prototyping tool for analysis. Our use of “operators” (discussed in 3.3) borrowed from the use of transformation techniques in skeleton based approaches, albeit our focus is on the use of object-oriented techniques.

The integration of patterns in Problem Solving Environments for Grid application development (e.g. Grid-aware workflow based PSEs), and their availability and manipulation by operators at all development stages (including reconfiguration support) is therefore, in our opinion, an important contribution on Grid usability.

3

Characteristics of the Model

Contents

3.1	Introduction	52
3.2	Pattern Templates	57
3.3	Operators	68
3.4	Summary	80

This chapter describes the main characteristics of the model and its components, namely Structural and Behavioural Patterns and their associated Operators; explains the way these components are related; and describes the methodology associated to the model.

3.1 Introduction

This first section in the chapter describes the major concepts that form the basis for the model. The subsequent sections in this chapter explain the model's entities, which are classified according to those concepts, and the way they are related. Each entity is characterised in terms of its semantics and applicability.

The basic concepts underlying the model are *Patterns* and *Pattern Operators*. In our model, Patterns capture commonly occurring structural and behavioural aspects in component composition, which can be typically perceived in distributed and Grid computing applications. As well as components in Component-oriented models, Patterns are first class entities, namely, patterns identified at design time are still present at execution time. Operators, in turn, manipulate those structure and behaviour aspects in a constrained way for configuration and reconfiguration of applications and for execution control.

Patterns and operators are divided and classified according to the above two dimensions; namely, the model provides both Structural and Behavioural Patterns and Operators. This separation of concerns is also a distinctive characteristic of the model as a way to promote flexibility:

- allowing different behaviours to be combined upon the same structure;
- reusing the same behaviour for different structures;
- allowing the refinement of the structure independently of the behaviour;
- changing the behaviour independently of the structure.

Finally, the last concept concerns a (suggested) methodology which can be associated with the model. Specifically, the persistence of patterns through the entire application life cycle, and their uniform manipulation by different operators at different stages, allows the definition of a systematic process for application configuration and its execution control. Such systematisation aims at helping expert users develop applications, or the methodology may guide less experienced users in that process. A basic methodology is explained in section 3.1.3, whereas extensions to that basic methodology, e.g. concerning reconfiguration, are deferred to Chapter 5. Nevertheless, we emphasise that these procedures are to be seen as possible guidelines for application construction and modification.

3.1.1 Structural and Behavioural Patterns

Structural Patterns (SPs) encode component connectivity, and identify common ways in which components may be combined within a given application domain. Structural constraints may be useful, for example, to represent common software architectures in high-performance or distributed computing applications. For instance, one of the uses of the data flow pipeline, in particular, is rendering, which involves a data input, simulation/rendering, and visualisation pipeline. Structural patterns may also contain a hierarchy, allowing the embedding of a pattern within another, these embeddings also being supported through specialised operators.

Structural Pattern Templates (S-PTs) consist of *component place-holders (CPHs)* to which individual components are instantiated at runtime, and a specification of the connectivity between those place-holders matching an associated pattern's semantics. Therefore, Structural Pattern Templates are instances of Structural Patterns in the sense that they are formed of a limited set of elements (i.e. CPHs) obeying the correspondent specific structural restriction. The number of those elements defines the *cardinality* of the *S-PT*. Structural Pattern Templates may include other S-PTs as their elements, forming a *Hierarchical S-PT*, and where each embedded S-PT is still directly accessible. In general in this work, the cardinality of a Hierarchical S-PT is considered to be simply the number of elements (either CPHs or S-PTs) in the first level of the hierarchy.

Behavioural Patterns (BPs) encode useful and commonly required functionality such that components within a Behavioural Pattern primarily identify interaction constraints, and not the exact functionality required from each individual component. As such, *Behavioural Patterns* can capture temporal or flow dependencies between components. Flow dependencies model data and control flows, and encode execution ordering on components (flow dependencies are typically used to express synchronisation constraints). *Behavioural Patterns* may be defined to specify: a) typical interaction models between components (e.g. Peer-to-Peer or Client/Server); and b) schemes which are used to update/change the behaviour of each component (i.e. new inter-dependencies are defined within the involved components).

Behavioural Pattern Templates (B-PTs) define the necessary set of actions/rules to support the semantics of the associated *Behavioural Patterns*, and characterise the role of each participant. These participants are defined as component place-holders/wrappers to be instantiated/applied to specific executable components.

The combination of *Behavioural Pattern Templates (B-PTs)* with *Structural Pattern Templates (S-PTs)* defines a *generic configuration*. This configuration is named a *Template Configuration* and represents a composition of *Pattern Templates*. Specifically, each individual *Pattern Template*, also designated as *SB-PT*, results from combining a particular *Structural Pattern Template (S-PT)* with one or more *Behavioural Pattern Templates (B-PTs)*. In turn, the binding of component place-holders within the above *Template Configuration* to specific executables defines a *particular application configuration*. In particular, an individual *SB-PT* when fully instantiated to executable components produces a *Pattern Instance (PI)*. Consequently, the final application's instantiated configuration consists of one or several combined *Pattern Instances (PIs)*.

Figure 3.1 describes the relations between the entities described above. Namely, a *S-PT* is created from a *SP*. Likewise, a *B-PT* enforces the semantics of a *BP* upon the involved elements. The combination of a *S-PT* with one or more *B-PTs* results in a *SB-PT* that following the instantiation of its component place-holders with executables, generates a *PI*. As can be observed in the Figure, a *PI* can also be obtained by first binding the component place-holders within a *S-PT* to executables, and subsequently combining the resulting *Component Instantiated Structural Pattern (CISP)* with one or more *B-PTs*. Finally, *SB-PTs* and *CISPs* have no direct relation between them – they only represent intermediate entities on the two distinct paths that lead to the generation of *PIs*.

The next sub-section describes the available operators to manipulate the above patterns.

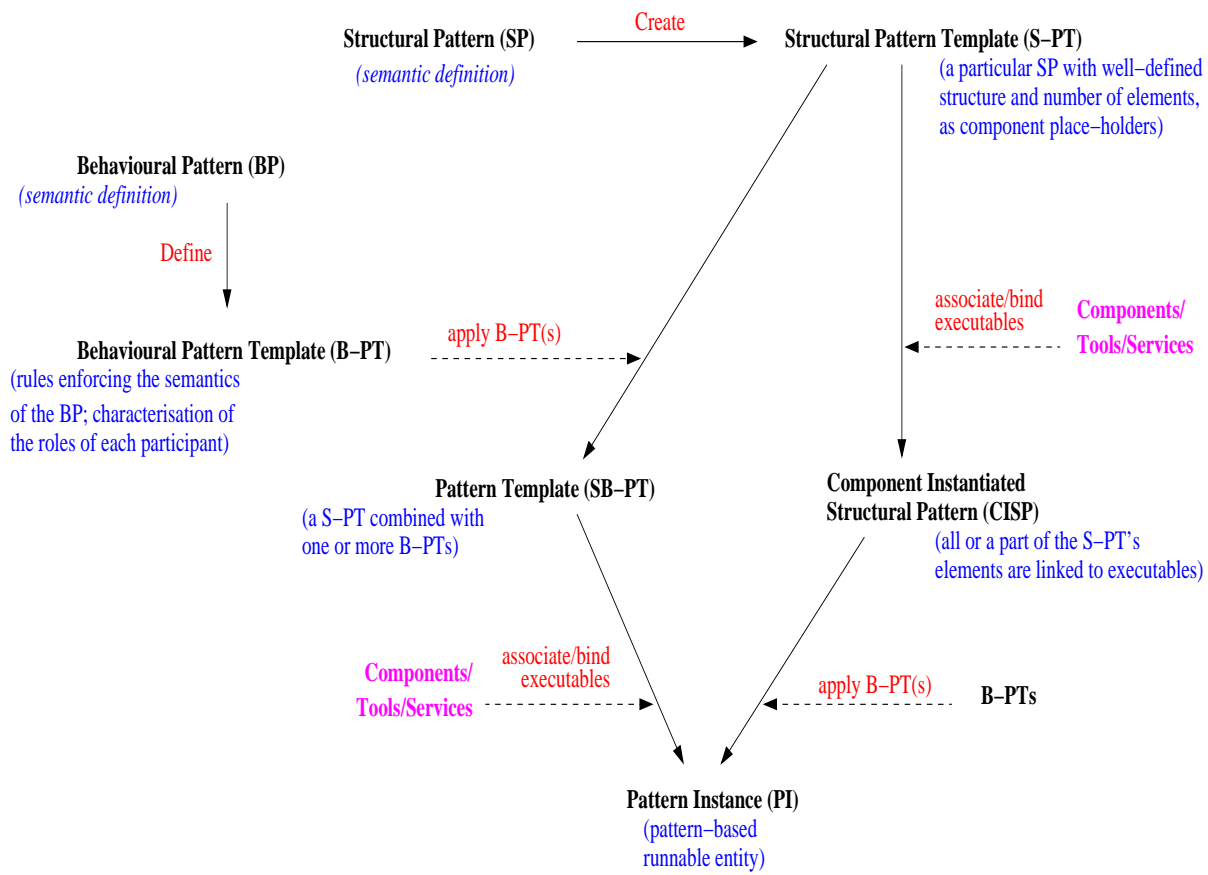


Figure 3.1: Relating the used pattern definitions.

Examples on their manipulation are presented along Chapters 4 and 5.

3.1.2 Structural and Behavioural Operators

Structural Operators manipulate *Structural Patterns* (either plain *S-PTs*, or the structure underlying *SB-PTs*, *CISP*, or *PIs*) enabling a constrained way to modify these patterns with the guarantee of preserving the distinctive structural characteristics of the manipulated patterns. Those operation constraints are defined by the semantics of each Operator, and relate to the result generated after the operator has been applied.

Behavioural Operators are used to configure *Behavioural Patterns*, and provide the user with execution control and reconfiguration capabilities. Similarly to Structural Patterns, each Behavioural Operator's semantics also define a constrained way of operating the involved patterns.

Although the actions of both Structural and Behavioural Operators are generally to be described in terms of *Pattern Templates (PTs)*, some are in fact applicable to *Pattern Instances (PIs)* only. This is the case of one of the operator categories described next, namely *Execution operators* which will be further discussed in section 4.4.

Operator Categories

Structural and Behavioural Operators are further classified into six categories, namely *Structuring*, *Grouping*, *Inquiry*, *Ownership*, *Execution*, and *Global Coordination*.

The first two categories include operators to define and change the structural connections between patterns (either pattern templates or pattern instances) building the application's basic configuration.

Inquiry operators, in turn, grant the user the possibility of evaluating the properties of different Structural PTs before applying the Structuring and Grouping operators. Inquiry operators allow the assessment of structural relationships between patterns (e.g. if one PT is part of another pattern), and may be used to analyse pattern compatibility (which includes both structural and behavioural properties). Finally, Inquiry operators also evaluate a user's access rights to manipulate a pattern which are defined through Ownership operators.

Ownership operators include operations to define who is entitled to access or manipulate a specific pattern, and also to associate specific actions to this pattern defined as being under the responsibility of a particular user or set of users.

Execution operators support the manipulation of *Pattern Instances (PIs)*, both for execution control and reconfiguration purposes, providing the user with operations for automating repetitive or periodic actions and to control the applications' evolution in face of changing resources or requirements. The user may in this way add or replace existing patterns in order to exploit such dynamic changes.

Finally, Global Coordination operators establish/change the behavioural dependencies between elements within a pattern according to some Behavioural Patterns or coordination rules.

3.1.3 The Basic Methodology

This section describes a basic methodology for the construction and control of *pattern-based application configurations*.

We define an application configuration as "pattern-based" if it results from the refinement/composition of Structural Patterns combined with Behavioural Patterns. Although the addition of individual components and direct connections between any elements within the structure are also possible and may be necessary for application configuration, this work focus on pattern manipulation towards application construction and control. Therefore, none of the discussions here presented makes any assumptions or identify the implications of establishing connections/dependencies among elements, which are outside the context of the defined patterns.

Figure 3.2 depicts the sequence of steps forming a basic methodology for the construction and control of pattern-based application configurations. This description assumes the existence of a supporting tool with an environment such that the user launches, for example, a *Problem Solving Environment (PSE)* editor (as, for example, identified in [18]) to connect components together, and that provides interfaces for execution control. For example, through a visual editor the user may select and build the necessary patterns, make other necessary direct connections between components, and apply the available operators. However, a visual-based pattern composition and manipulation is not mandatory as the defined methodology steps and

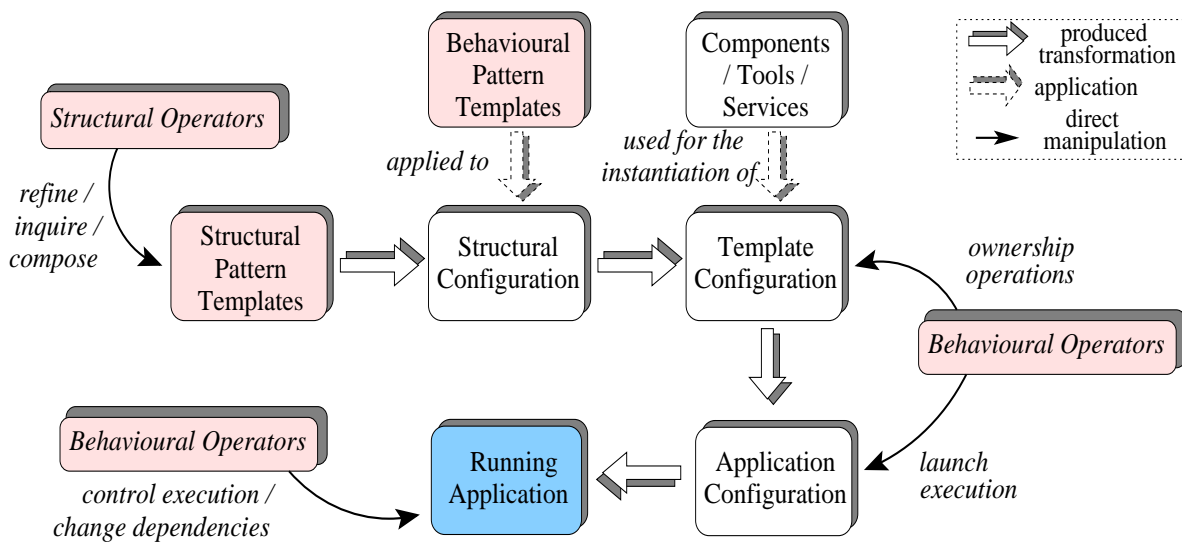


Figure 3.2: The basic steps of the methodology.

the definition of operator sequences can also be achieved through scripting languages. Such scripts, in turn, may provide a systematic way of building those application configurations and controlling their execution.

The methodology steps represented in Figure 3.2 correspond, in part, to the left branch of Figure 3.1, namely, the steps illustrate: the generation of *Structural Pattern Templates* (S-PTs) from *Structural Pattern* (SPs) definitions; their combination with *Behavioural Pattern Templates* (B-PTs) leading to *Pattern Templates* (SB-PTs); and finally, the instantiation of SB-PTs to *Components/Tools/Services* generating *Pattern Instances* (PIs).

Methodology Steps

1. The user selects, typically from a repository, the most adequate Structural Patterns generating Templates (i.e S-PTs – *Structural Pattern Templates* in the Figure 3.2) which enable the static composition of components and over which the (data and control) flow dependencies will be defined.
2. The user combines and refines the selected Structural Pattern Templates by applying the available Structural Operators with the guarantee that these are invariant regarding the target pattern template structure. Structural Operators of the Structuring, Grouping and Inquiry categories are available from a repository for user selection. At any time, the user can also modify the structure of a Pattern Template directly by using the editor commands.

The pattern refinement process may be associated with the selection of further necessary Structural Patterns until the desired *Structural Configuration* (in the Figure) is produced (which may be a single S-PT or several combined S-PTs). At this stage, the designed configuration does not restrain the flow dependencies between the involved elements in any way. Meanwhile, new Structural Patterns may be defined and saved in the repository for further reuse. Furthermore, the generation of Pattern Templates as well as the application of sequences of Operators may be triggered by running user defined scripts.

3. The user specifies the behavioural dependencies between the elements by first selecting, from a repository, and then applying, the suitable *Behavioural Patterns* over the previously defined *Structural Configuration*. The result is a *Template Configuration* (i.e. a combination of *SB-PTs*) with defined data and control flows between the elements. These *SB-PTs* may be statically configured using some *Behavioural Operators*, for example, for the definition of ownership restrictions. The *Template Configuration* can also be saved for later reuse. To produce the final *Application Configuration* in the Figure 3.2 (i.e a combination of *PIs*), the user instantiates the component place-holders with the necessary runnable components, tools, or services. This instantiation consists of relating the elements within the *Application Configuration* to the necessary information to run the executables or to access the selected services.
4. The user launches the application's execution and configures its run-time behaviour through *Behavioural Operators*. Sequences of *Behavioural Operators* may also be invoked from user defined scripts.

The availability, at run-time, of *Structural* and *Behavioural Patterns* as manipulable first-class entities just like the executable components (e.g. accessible through a PSE editor), provides the user with a higher level of control over the application. Through *Behavioural Operators*, the user may manage the application's execution as a whole, or may restrict the configuration/control of the behaviour to just one or more *Patterns*. Moreover, further modifications to the application may be imposed, either statically or dynamically.

The division into these four stages of design is inspired on existing user scenarios for application construction in Problem Solving Environments. Based on our approach, a user must first commit to a *Structural Pattern*, and then to a *Behavioural* one. *Structural Patterns* may, for example, try to capture how many machines (for instance) or groups are necessary to execute a given application – and do not instantiate these to particular instances until the *Behavioural Operators* are applied. The four stage approach in this example reflects the approach adopted by application schedulers – but tries to abstract this as a collection of patterns and operators – and brings it closer to the application construction process.

The following sections describe the selected *Structural* and *Behavioural Patterns*, introduce a relevant set of *Operators*, and outline the capabilities supported by the model. However, the existence of different entities, i.e. a set of *Structural* and *Behavioural Patterns* leading to different associations, and their different manipulation through the entire life-cycle, ultimately suggests a high number of combinations to be discussed that it was not possible to describe exhaustively in the context of this work.

3.2 Pattern Templates

This section describes the selected *Structural* and *Behavioural Pattern Templates* (*S-PTs* and *B-PTs*) that aim to represent frequent configurations and behaviours present in distributed and Grid environments, at the level of the components integrated in Problem Solving Environments in particular.

The chosen set of *Patterns* can be seen as identifying a small *Pattern System* which was found to provide adequate expressiveness for the collection of application examples that were

studied. New Patterns may be added if found required for specific applications, as the model is amenable to extensions.

3.2.1 Structural Pattern Templates: Topological

Topological patterns represent structures that frequently occur in distributed and Grid systems. For illustration purposes, we identify three basic architectural layouts as possible candidates to include in the model within this category: star, pipeline, and ring.

Star Pattern

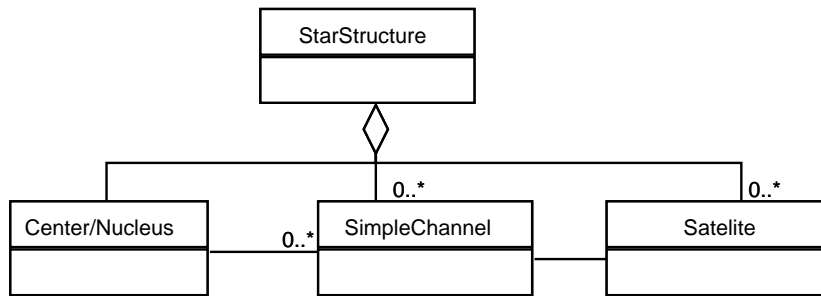


Figure 3.3: *The Star pattern.*

The *Star* pattern (see figure 3.3 for the UML [68, 69] definition) is an aggregation of three components: the *Nucleus* is the center of the star; a *Satellite* represents an element communicating with the star; and the *SimpleChannel* binds together a *Satellite* to the *Nucleus*. The *Nucleus* may be connected to several instances of *SimpleChannel*, but each *SimpleChannel* is only connected to a single *Satellite*. The star topology defines the general interaction of several common Grid/distributed applications. For example, in the Client/Server model, the clients accessing the service can be structurally represented by the satellites with the server standing at the nucleus of a star. Likewise, the Master/Slave model can also be structurally represented by the star architectural layout.

Pipeline Pattern

The *Pipeline* pattern is a sequence of stages which communicate with each other. The pattern occurs frequently in many applications. For example, a scientific application produces data to a sequence of filters (like *Data Analysis Tools*), and the pipeline is terminated in a *Visualisation Tool* where the user can follow the application's execution. The pattern's structure (as shown in Figure 3.4) was adapted from the *Pipes and Filters* pattern [10]. The structure can be generally represented by three components (see Figure 3.4): a *DataSource* produces data to a *Connector*, and the *DataSink* consumes data from the *Connector*.

The *Connector* has a recursive structure, as illustrated in Figure 3.5. A *Connector* may be a *SimpleConnector* (similar to a Unix pipe, an event channel, a data stream channel, etc.) or it

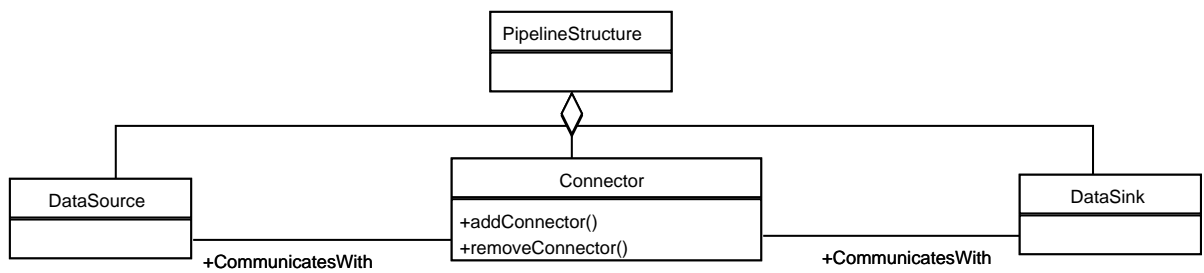


Figure 3.4: *The Pipeline pattern.*

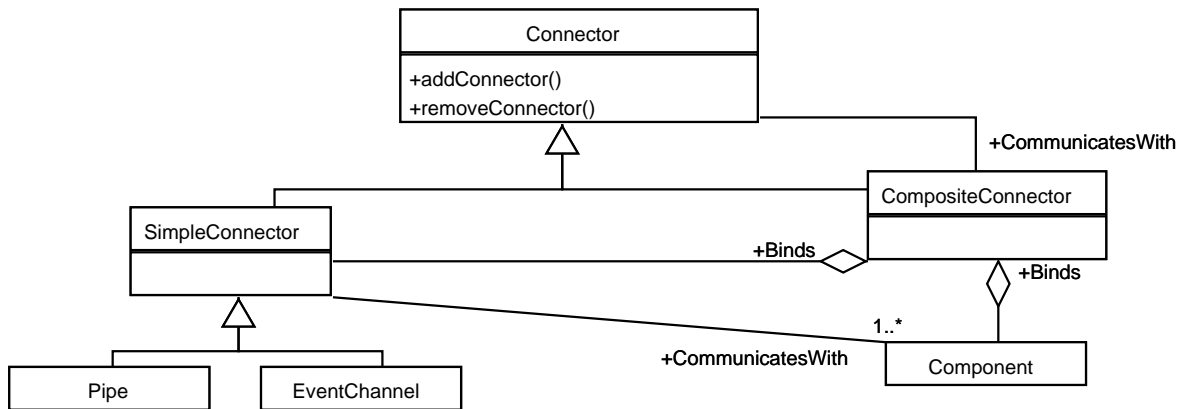


Figure 3.5: *The Connector pattern.*

may be a *CompositeConnector*. The latter is a connected association of a *SimpleConnector* and a *Component*. Recursively, the *CompositeConnector* may be connected to another *Connector* and the recursion terminates at the *SimpleConnector*. The number of recursions define the number of necessary associations (each comprising a *SimpleConnector* connected to one *Component*) which will define the pipeline stages.

Ring Pattern

The *Ring* pattern represents, like the pipeline, a sequence of stages, but with no "first" or "last" stage. The structure of the *Ring* pattern (Figure 3.6) is also based on the *Connector* structural definition. The difference is that every *Component* is always connected to two *SimpleConnectors* (in the limit, the unique component will have two connections to a single *SimpleConnector*). The number of recursions in the *Connector* definition will define the number of stages in the ring. For simplification purposes, the definition of the *Connector* is not complete in Figure 3.6 (see Figure 3.5 for its complete definition). This topological structure can be found in a number of applications, both in the context of application execution (such as for modelling interactions within a local area network) to logical topologies such as supporting an authentication chain when approving participants with multiple certificate servers. Each server delegates an authentication request to the next domain, and the last server replies to the original client. This chain based mechanism can also be found when resolving the address/location of an executable us-

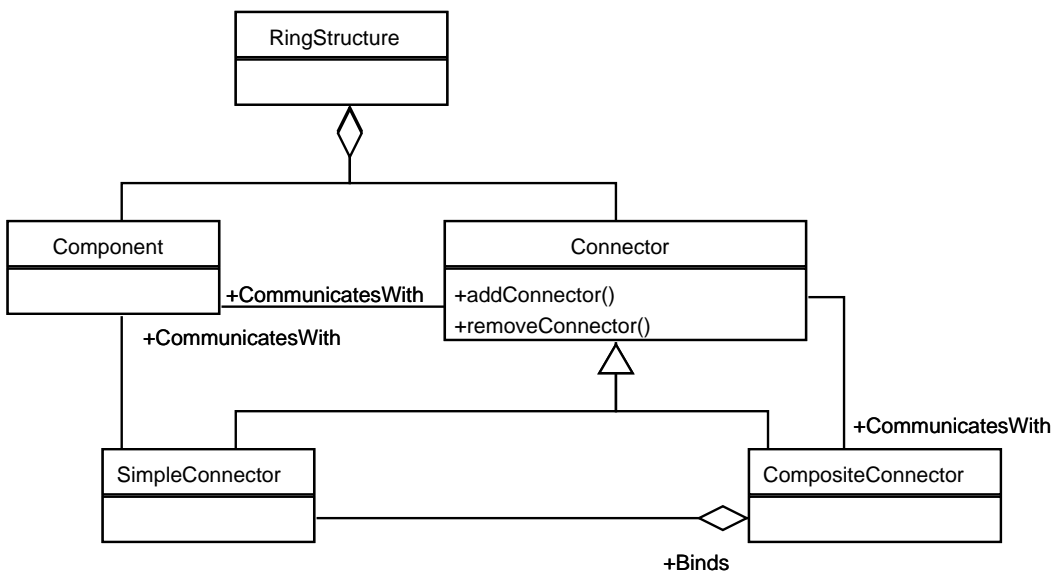


Figure 3.6: *The Ring pattern.*

ing a directory lookup service (as found in the Globus MDS [11]).

Although the star, pipeline and ring topologies represent, by themselves, the underlying structure of many distributed applications, those topologies can be combined to configure other common topologies in distributed systems (this will be exemplified in Chapter 7). Moreover, the benefits of topology-awareness for distributed and Grid computing are presented in [210], for example, for reducing communication costs. As such, application developers may benefit from pre-defined topologies for application configuration.

3.2.2 Structural Pattern Templates: Non-Topological

The *Adapter*, *Facade*, and *Proxy* design patterns (adapted from [9]) are examples of non-topological Structural Patterns that are particularly useful in the context of distributed and Grid computing.

Adapter Pattern

The *Adapter* pattern allows communication between two elements when they do not have the same interface (see Figure 3.7 for the UML definition). In a Grid environment, the *Adapter* pattern has applicability, for example, in the adaptation of services, or as wrappers for legacy codes (such as Fortran binaries). If the client is expecting a different interface from the one provided by the server, the adapter can act as a translator. This pattern is also particularly useful for providing a mapping between the interface of an existing code and a pre-defined component data model for Grids, such as CCA [27].

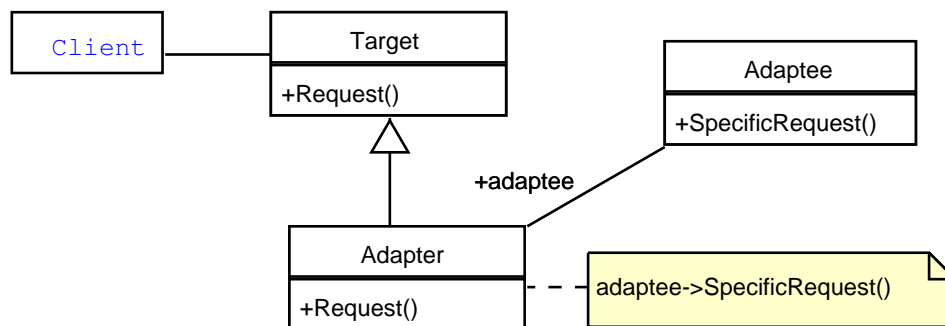


Figure 3.7: The Adapter pattern [9].

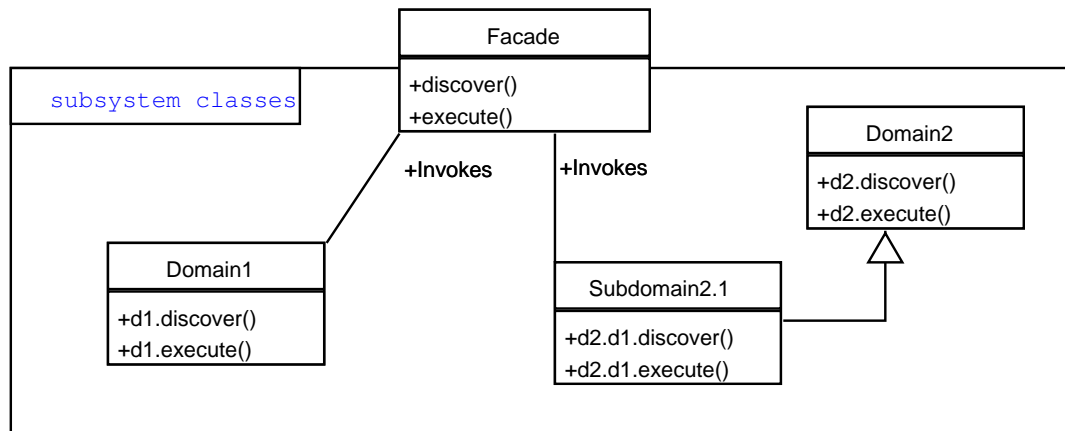


Figure 3.8: The Facade design pattern [9]. Example: the "Facade" provides a unified interface for accessing domains in the Grid environment, redirecting the calls to services like "discover" and "execute".

Facade Pattern

The *Facade* pattern (Figure 3.8) is useful when a system may be divided into several sub-systems, and the access to communication/entry-point into the system needs to be restricted. The *Facade* pattern occurs in the structuring of the Grid into multiple "domains". The access to each domain (sub-system) in the Grid may be via a *Facade* interface.

Proxy Pattern

The *Proxy* pattern is also frequent in distributed systems. The access to Grid services, for example, is usually achieved through a proxy (or gatekeeper). The structure of the pattern (Figure 3.9) consists of an abstract interface (the *Subject*) representing the service, the implementation of the service (*RealSubject*), and a surrogate (*Proxy*) which forwards the request to the implementor of the service.

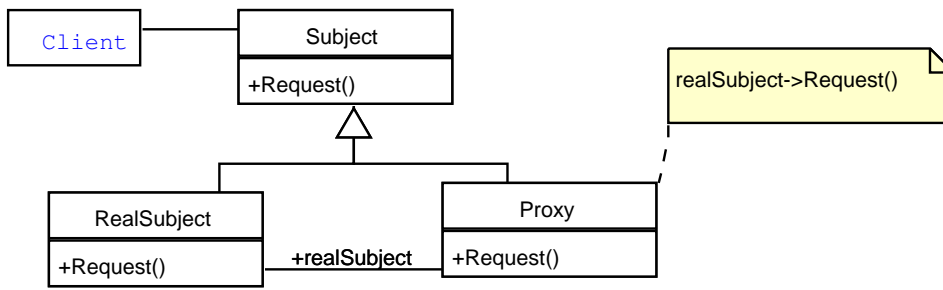


Figure 3.9: *The Proxy pattern [9].*

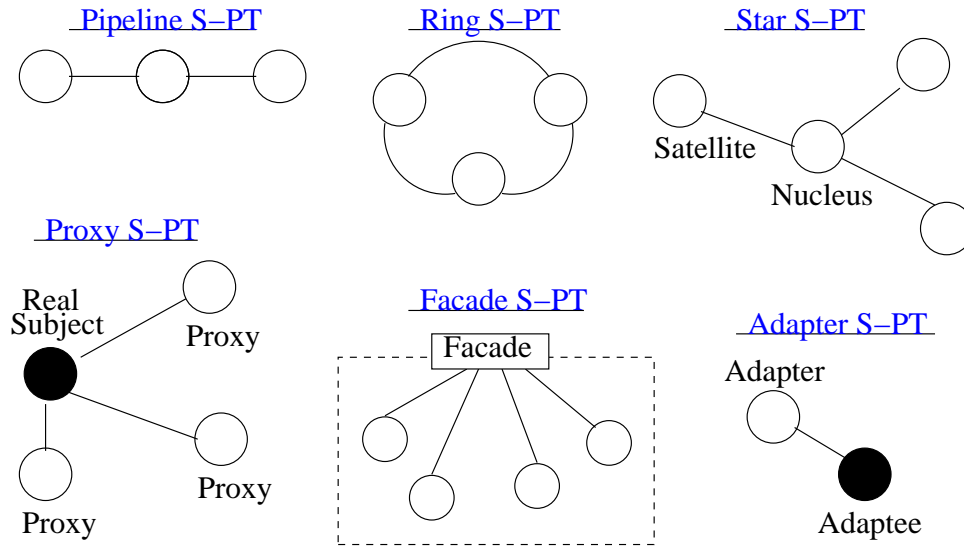


Figure 3.10: *Graphical representation of examples of Structural Pattern Templates (S-PTs).*

3.2.3 Graphical Representation of Structural Pattern Templates

The graphical representation of instances of the above mentioned Structural Pattern Templates (S-PTs) is shown in Figure 3.10. Such representation is used throughout the thesis, both in the visual description of the Operators' semantics and in some examples in Chapter 7.

The basic elements in the Structural PTs' representations are: a) the component place-holders, which are depicted as circles or as a small rectangle in the case of the Facade PT's main element; and b) the associations between the place-holders defining the structure, which are depicted as solid lines connecting the circles (or rectangles).

Figure 3.10 illustrates: one Pipeline and one Ring templates (S-PTs) each with three elements (i.e. the *cardinality* of both S-PTs is equal to three) ; one Star and one Proxy templates with three satellites and proxies, respectively (here the cardinality of the S-PTs is equal to four elements) ; one Facade S-PT with four component place-holders to be instantiated to four sub-systems (i.e. the cardinality of the S-PT is equal to five); and an Adapter template (the cardinality of the S-PT is equal to two).

The *visual representation of the topological Structural Patterns* consists of similar component place-holders indicating that there are no restrictions on the type of components or services

which will instantiate them. Compatibility evaluation of two connected place-holders will be undertaken at instantiation time. For example, in the Triana tool [40], the data-flow connection between two consecutive services in a topological Structural Pattern is only allowed in case of data type matching between the source's output and the destination's input data.

The *visual representation of the non-topological Structural Patterns*, in turn, distinguishes in some way the component place-holders within a template (e.g. the "Facade" element within the structure of the "Facade S-PT" in Figure 3.10 is represented differently from the sub-system elements). This aims to highlight the fact that each element within each non-topological Structural Pattern has associated structural constraints that have to be met when instantiating those component place-holders to specific executable components.

3.2.4 Behavioural Pattern Templates

Behavioural Pattern Templates (B-PTs) capture recurring themes in component interactions defining their (control and data) flow dependencies to be described through rules or schemes.

Specifically, the patterns described here aim to represent common component interdependencies appearing within Grid applications, in particular, and in distributed systems in general.

Master/Slave Pattern

One of the selected patterns is the *Master-Slave* [10] which is commonly found in distributed and parallel systems. The pattern comprises two types of elements, namely, a master, and a set of "N" slaves. The work is divided by the master into sub-tasks that are performed by the (usually independent) slaves. Afterwards, the partial results returned from the slaves to the master are used by the latter to compute the final result. The Master-Slave pattern is present in grid-enabled environments such as XtremWeb [221], BOINC [222], the Condor Master Worker project [224], and Nimrod/G [223].

Client/Server Pattern

Another common pattern is the *Client-Server* [10, 220] where a particular element, the server (either centralised or distributed), processes the requests made by concurrent clients. This pattern is similar to the Master-Slave although in the former the control flow is more complex. The Client-Server pattern can be identified in innumerable distributed applications and architectures, including Grids. For example, Client-Server based systems such as NetSolve [225], ICENI [227], and Ninf-G [226], provide access to several Grid resources.

Peer-to-Peer Pattern

The *Peer-to-Peer* pattern [220], in turn, eliminates the difference between "server elements" and "client elements" – all components within a Peer-to-Peer system function simultaneously as "clients" and "servers" to the other components. Consequently, whereas in a Client-Server system the interaction is undertaken through a central server which is prone to resource bottlenecks, Peer-to-Peer systems support sharing of computer resources (e.g. data, storage, CPU cycles) by direct exchange between the involved components and are also more resilient to

failures. The Peer-to-Peer model became very popular in distributed applications and systems [230–233] including for the access to Grid resources [21,234,235].

Producer/Consumer and Streaming Patterns

Another pattern which is also common in distributed systems, is the *Producer-Consumer* pattern [236]. This pattern captures the coordination of the asynchronous production and consumption of information, where the data flow is unidirectional: from one or more producers elements to one or more consumers elements. The Producer/Consumer pattern is useful to decouple entities that produce and consume data at different rates (and commonly data is to be processed in order), and hence data is buffered between the producer and the consumer. The Producer/Consumer model is, for example, frequently used in Monitoring Services within Grid environments [240,242].

The *Streaming* pattern is a variant of the Producer/Consumer and represents a continuous production of data. This pattern is used in this work to represent both the data flow production characteristic of scientific calculations and tools (which may present a high throughput), as well as the streaming characteristic to audio and video media.

Parameter-Sweep Pattern

As for the *Parameter-Sweep* pattern, it represents the repeated invocation of a component with unique sets of input parameters – the same code is run multiple times and a single parameter varies over a range of values or multiple parameters vary over a large multidimensional space. The Parameter-Sweep pattern is used as the basic execution model in many scientific and engineering applications (e.g. applications based on Monte-Carlo simulations or parameter-space searches), and due to their large-scale and loosely coupled nature, these applications are well suited for the Grid. The Parameter-Sweep pattern can be found in systems such as the Apples Parameter Sweep Template (APST) project [228] and Nimrod [19].

Observer/Publish-Subscriber Pattern

Considering the need to manage the consistency among a set of related, although decoupled, entities, we have selected the *Observer/Publish-Subscriber* pattern [9,10] as one important Behavioural Pattern for distributed/Grid systems. In this pattern, one or more entities may register themselves (or be registered) to be notified when a certain event occurs at an observed entity. The latter is called the *subject* and the former are called *observers*. The pattern defines a one-to-many dependency between the subject and its observers, and the notification may either simply raise an event to be notified at the observers, or it may carry information concerning the change at the entity being observed.

For instance, the Observer/Publish-subscriber is used in many Graphical User Interface (GUI) applications (e.g. the Java Swing GUI itself uses the pattern), and applications where changes to some (centralised) data have to be notified to the (distributed) observers, like applications providing information about stock quotations, or health care applications that monitor the patients' status. The Observer pattern is increasingly being used in a Web services context

(e.g. [238], and in [239] the pattern is used as the basis for a Web Services Notification family of specifications), and in the Grid context, the pattern is used, for example, in support of collaborative visualisation of scientific applications [237].

One final remark, concerns the possibility of making the distinction between the *Observer* pattern and the *Publish/Subscriber* pattern. The former may be considered simpler than the latter in the case that the subjects to be observed (e.g state, events) pertain to a single entity, and that this one keeps a registry defining which other entities are observers, and which subjects these observers are interested on. The *Publish/Subscriber* pattern, on the other hand, decouples the entities that possess the observable subjects (e.g. events) and the registry. This registry may perform, for example, the following tasks: a) to provide information of whose subjects are observable, and who provides them; b) to register who is interested on which subjects; c) to relate the entity that provides the observable subjects, and the entities that are registered as their observers. Nevertheless, the examples presented in this work consider, in general, the (simpler) *Observer* semantics, and not the more complex *Publish/Subscriber* semantics.

Service Design Patterns for Grids

Concerning patterns specific for Grids, several were first described by Walker and Rana in [34]. For example, the *Service Adapter Pattern* is a Behavioural Pattern that captures the properties/behaviours which are necessary to provide an application as a service, and the *Service Migration Pattern* enables the migration of a service to another computational platform (which provides enhanced computational resources).

Mobile Computing Patterns

Finally, we elected three general patterns from the mobile computing domain [243] as relevant to be part of a repository of Behavioural Patterns, namely *Code-on-Demand*, *Remote Evaluation*, and *Mobile Agent/Itinerary*. The mobile code paradigm represents a decoupling between the behaviour of distributed components from their location, and therefore it is an important concept both for distributed systems and Grid computing [246].

– Remote Evaluation

The *Remote evaluation* [243] scheme generally represents the sending of code, by a client, to be executed by a remote server that, afterwards, returns the results back to the client. This scheme is fundamental in Grid computing, for example, as a way to provide clients with the capability of delegating code execution to high performance computational servers, or to perform data processing code on servers that have local access to large amounts of data, therefore avoiding expensive data shipping. Ninf [251], NetSolve [225], and Globus [249, 250] are examples of Grid environments supporting remote evaluation, and some already provide higher-level abstractions for remote submission of jobs [247, 248].

– Code-on-demand

The *Code-on-demand* [243] scheme, in turn, represents the sending of executable programs by server computational units to be run by other client computational units. This scheme is the basis of the Java applets [244] model, and it is also present in Grid environments (e.g java applets in [249] support the runtime steering of the applications through a friendly GUI; the "Coglets

framework” [245] aims to support arbitrary execution environments, namely Java applications running applets).

– Mobile Agent/Itinerary

The third pattern belongs to the Mobile Agents domain [243]. A mobile agent is a complete software component, which includes code and also state (contrary to the two previous patterns), that moves between different execution environments within a network (e.g. for gathering information and negotiating with other agents on behalf of their clients).

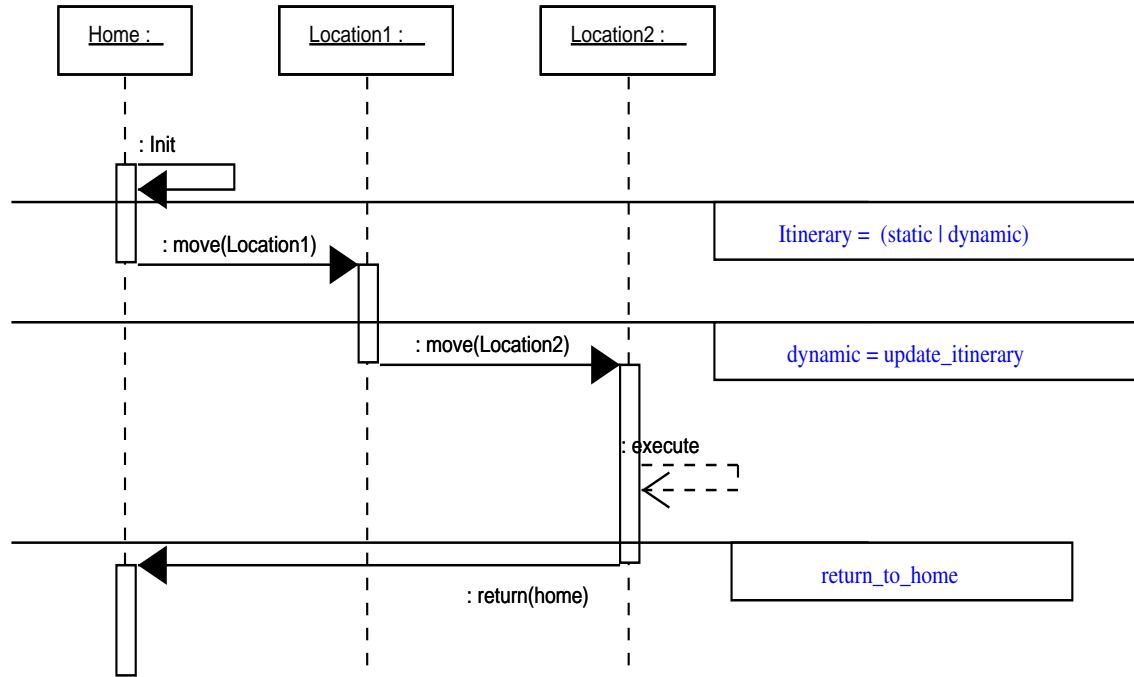


Figure 3.11: A Sequence diagram for the Mobile Agent/Itinerary pattern defining a possible itinerary for the component.

The selected pattern is the *Mobile Agent/Itinerary* [35] (identified as a "Traveling Pattern" within mobile agents), and Figure 3.11 illustrates its sequence diagram. In this pattern, a component is initialised at a given location (Home), and may move to another location based on a pre-defined itinerary or on a dynamic one.

The mobile agents paradigm and the *Mobile Agent/Itinerary* pattern in particular have been considered as a significant contribution for Grid environments [246, 252, 253, 255].

3.2.5 Combining Behavioural and Structural Patterns

This section presents some examples of Behavioural Patterns applied to Structural Patterns. Specifically, the combination of both classes of patterns results in specifying, i.e. annotating, the behaviour of each element/component within the Structural Pattern Template (*S-PT*) in conformity with one or more Behavioural Pattern Templates (*B-PT*). The role of each pattern element within the applied Behavioural Pattern(s) has a simple visual representation in all examples presented in this chapter.

We define two forms of applying a Behavioural Pattern to a Structural one:

1. All elements within a Structural Pattern will be coordinated according to a unique Behavioural Pattern.
2. Diverse Behavioural Patterns may rule the flow dependencies among different Structural Pattern's elements.

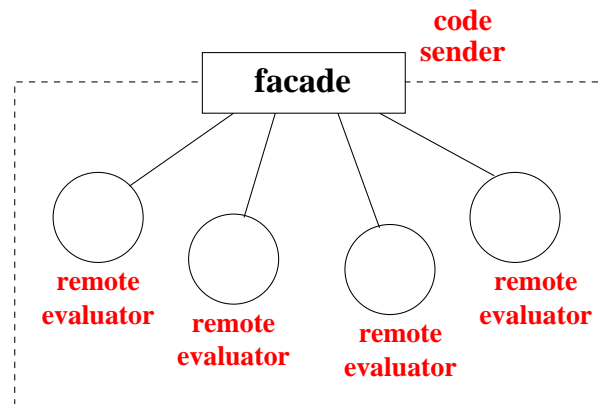


Figure 3.12: A Remote Evaluation B-PT applied to a Facade PT.

As an example of the first situation, Figure 3.12 depicts the combination of the *Remote Evaluation* Behavioural PT and the *Facade* Structural PT. The component place-holder representing the facade element is annotated as playing the role of the *code sender*, and all component-place holders to be instantiated with particular subsystems are tagged as *remote evaluators*. Upon instantiation of all place-holders their defined (flow and control) dependencies will enforce the *Remote Evaluation* behaviour. Such combination may be useful, for example, for interfacing data processing on different database systems. The facade component would therefore provide an uniform user interface hiding those (sub-systems) databases' specific interfaces through which users may submit specific executable code for searching/analysing/processing data. This code would be dispatched by the *code sender* (*facade*) to be remotely evaluated by the adequate database(s) .

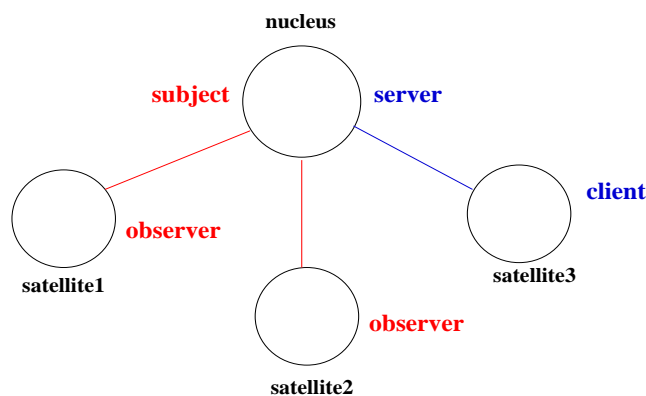


Figure 3.13: Two Behavioural Pattern Templates, namely Client/Server and Observer B-PTs, applied to a Star PT.

An example of the second situation, namely different Behavioural Patterns Templates (B-PTs) applied to the same Structural PT, is shown in Figure 3.13. A *Star* Structural PT is combined

with: a) an *Observer* B-PT, where the "nucleus" is annotated as the *subject*, and "satellite1" and "satellite2" are component place-holders annotated as its *observers*; and b) a *Client/Server* B-PT where the "nucleus" is tagged as the *server* and "satellite3" is the *client*.

Such combination of structure and behaviours can be adequate, for example, for distributed health care monitoring systems. Typically, the structure may represent the connection of remote monitors ("satellites") to the central monitor ("nucleus") in a patient's room that is responsible for collecting and providing information about the patient's vital signals. One remote monitor may be registered to simply observe heart beat frequency and brain activity values assisting the day resident nurse. Another nurse in charge of daily medication may rely on information provided by a second remote monitor registered to be notified of parameter values such as blood pressure. The behaviour of those two remote monitors may be therefore ruled by the *Observer* pattern guaranteeing automatic data notification, and the tools representing the two monitors would, for example, instantiate the "satellite1" and "satellite2" elements within the Star SB-PT in Figure 3.13.

However, a third monitor might also be used to inspect the patient's vital signals on demand. This monitor might support, for example, a doctor that only needs to inquiry about the patient's condition on a less frequent base and, therefore, a *Client/Server* orchestration is sufficient. The tool representing the doctor's monitor would, typically, instantiate the element "satellite3" in the same Figure.

Both Template Configurations (i.e. combination of Behavioural and Structural PTs still not instantiated to specific components/services) described in the two previous examples may also be reused in distinct application scenarios.

Having described the selected Structural and Behavioural Pattern Templates, the next section presents a set of *Operators* for Pattern Templates' manipulation in order to build architectures and to control the execution of the final configuration.

3.3 Operators

Operators enable constrained manipulation of patterns by a developer or an execution controller, and provide a limited set of methods to achieve this. Operators provide transformations between patterns, albeit subject to a set of constraints, with the guarantee that the semantic restrictions of the manipulated patterns are kept consistent with the definitions presented in section 3.2. All manipulated patterns are uniquely identified by their name, which is used as an argument to the applicable operators, provided that those pattern's types are conform to the type of parameter defined for the operator.

Two kinds of operators exist within our approach: *Structural Operators*, and *Behavioural Operators*. Operators are further divided into six categories, namely the *Structuring*, *Grouping*, *Inquiry*, *Ownership*, *Execution*, and *Global Coordination* categories.

This section first describes the meaning of each category (sub-section 3.3.1) followed by a brief summary of the semantics of the available operators within each category (sub-sections 3.3.2 to 3.3.6). A full operator semantics will be described in Chapter 4, but only for the *Structuring*, *Grouping*, and *Execution* operators. In Chapter 4 the operators' applicability, in

particular, will be described in terms of individual operators, as well as in terms of operator sequences (with a particular ordering) involving sets of (different) operators. Those particular sequences include operators belonging to the same or different categories.

Operators may as well be combined, leading to "compound operators", although this is only allowed if operators from the same category are chosen – to ensure consistency of the result. Implementation-wise, each operator category can be considered to be implemented as a separate class library, and each operator is a method call within the library, but of course, the model definition is neutral regarding the concrete implementation.

3.3.1 Operator Categories

Structuring: These operators are used to establish or modify the connectivity between components in patterns.

Grouping: Operators to support grouping allow patterns to be combined and subsequently be manipulated as a whole. Grouping also involves including patterns within each other, thereby assisting on building hierarchies. Consequently, a *Hierarchical Pattern* consists of a pattern which contains at least another pattern as one of its members.

Inquiry: Inquiry operators support comparison between pattern templates, to check for consistency or compatibility (for instance). Inquiry operators may also be used to verify structural or behavioural properties associated with a template, and return a boolean value on evaluation.

Ownership: Ownership operators enable the modification and access rights of a template to be controlled. These may be used, for example, to define which user or users are allowed to manipulated a pattern (or a group representing a set of patterns). The owner of a pattern may also delegate access to a single user or group of users to modify the pattern.

Execution: Operators to control execution, for example, to assist the mapping between the application configuration within a Problem Solving Environment and a resource management system. Execution operators provide functionalities for managing execution of *Pattern Instances*, and provide support for changing the behavioural properties of pattern instances dynamically. Execution operators may connect to pre-defined scripts for starting, stopping, resuming, etc, component execution, or may be mapped to the protocol layer between a global Scheduler and local Schedulers (e.g. to reserve and allocate resources in the Grid). Specifically, the execution operators rely upon the functionality available within a resource management system, and depend on obtaining monitoring information. The mapping between the operators and the particular functionality of resource management systems therefore cannot be completely pre-defined. We therefore rely on an intermediate API (such as a global Scheduler as mentioned above) to enable these operators' execution.

Global Coordination: This type of operators support both the definition and the changing of the behavioural dependencies between elements within a pattern. These will rule those

elements inter-related behaviour at execution time. Such definitions and modifications may as well be applied upon executing *Pattern Instances*.

All the operators in the previous categories have a common parameter type which identifies the pattern(s) being operated. Such parameter is usually identified as “SP”, or “P”, “P1”, “P2”, etc., and it represents a particular pattern at a particular point of the development process, from design to execution time. Namely, the parameter may either represent a particular *Pattern Template (PT)*, *Pattern Template (SB-PT)*, *Component Instantiated Structural Pattern (CISP)*, or *Pattern Instance (PI)*. The argument that instantiates a parameter “P”, “P1”, etc., identifies uniquely that particular pattern within the system. Each pattern operator, as described in the following, takes one (or more) pattern(s) of the above types as input, and returns a pattern, a boolean result, or particular data pertaining to the operator’s semantics.

Some operators, in particular, also make reference to a particular member, or element, of the operated pattern. In fact, a pattern consists of a set of members/elements which are related according to that pattern’s semantics, and where each member has its own unique identifier within the pattern. Depending on the pattern’s type being manipulated, a pattern’s *member/element* may represent:

- a component place-holder, e.g. a member of a *S-PT*;
- an annotated component place-holder, i.e. a component place-holder with an associated role within a particular *Behavioural Pattern* (e.g. a member of a *SB-PT*);
- a component instantiated element, i.e. a component place-holder which was already bound to an executable (e.g. a member of a *CISP*);
- a runnable element, i.e. an element of a *PI*; namely, an element within a particular Structural Pattern for which the necessary (data and flow) dependencies were already defined, and which is associated to the particular executable to be run in that context.

The identifier of a pattern’s *member/element* is also sometimes named as *position*. Consequently, an argument named *position* identifies solely a particular member within the pattern. Furthermore, the reference to the *members* of a pattern usually alludes to all elements within that pattern at its top *level*, i.e. *level one*. Specifically, all members of a non-hierarchic pattern (i.e. a pattern which does not include any another pattern as its member), are said to be positioned in the *level one* of that pattern. On the other hand, a *Hierarchical Pattern* has two or more *levels* and one or more elements at its *level one* are also patterns.

Moreover, for the majority of operators (e.g. *Structural Operators*), their actions upon a specific *Hierarchical Pattern* have effect only upon the members in level one. This means that, in general, those operator’s actions are not recursive. Nevertheless, since inner patterns within a *Hierarchical Pattern* are directly accessible through their identifier, any operator can be directly applied to those inner patterns independently from the other members of the outer pattern. Typically, the identifier of an inner pattern consists of a name sequence (e.g. concatenation) which starts with the name of the outmost pattern, and ends with the name of the inner pattern. The discussion on the structural operation of *Hierarchical Patterns*, in particular, will be presented in section 4.3.

The next sub-sections summarise the available operators within each of the described categories, providing an introductory description of their semantics.

A more detailed description on some operators is discussed in Chapter 4, and generally in the context of *Pattern Templates (PTs)*. Nevertheless, some operators like the *Execution Operators* are only meaningful if applied to *Pattern Instances (PIs)*. Consequently, in Chapter 4, their general semantics is discussed in the context of *PIs*. Subsequently, operator semantics are again discussed in Chapter 5, namely, section 5.2 further clarifies the semantics of some relevant operators when applied in the particular contexts of *SB-PTs*, *CISPs*, and *PIs*; and the role of the different kind of operators towards reconfiguring an executing application is discussed in section 5.3.

3.3.2 Structuring and Grouping Operators

These operators are used to create *Structural Pattern Templates (S-PTs)*, and to modify their structure maintaining the structural constraints of the original PT, and include:

Create(SP, name [, nElems]) Creates a new instance of the *Structural Pattern* identified by “SP”. Consequently, a new *S-PT* is generated with a number of component place-holders defined by the parameter “nElems”. The parameter “name” is assumed to be a unique identifier of the *S-PT* within the system. The “nElems” parameter may be considered optional in case the structure of the *Structural Pattern* “P” implies a pre-defined number of elements (e.g *Adapter Pattern*).

Eliminate(P) The pattern identified by the parameter “P” is deleted and its identifier is removed from the system.

Replicate(n, P [, {id1, ..., idn}]) The pattern “P” is replicated “n” times, and these replicas become independent Patterns – each replica acquires its own unique identifier and can be individually operated. Although the generation of those unique identifiers is to be implementation dependent, an extra parameter to the operator is also defined for clarity purposes and it is used in our examples – the labels “id1”..“idn” identify the name of each of the replicas.

Replace(P1, P2) Replaces the pattern fully identified by “P1”, as a single entity, with the pattern identified by “P2”. For example, if “P1” is the identifier of a Star *S-PT* which is embedded in the first stage of a pipeline, and “P2” represents a Proxy *S-PT*, this pattern takes the place of “P1” in the first stage of that pipeline.

Reshape(P1, P2) A pattern P1 is transformed into pattern P2. This structural transformation is constrained by the structural restrictions of the original and destination patterns (“P1” and “P2”, respectively).

Increase(n, P) Adds “n” component place-holders to the *first level* of a pattern “P” according to its structural definition which establishes where the new elements are to be placed within that structure. Considering that the *Increase* has effect only on the first level of a pattern, such means that this operator is not recursively applied to the inner elements in

case “P” is a *Hierarchical Pattern*. Nevertheless, the *Increase* operator can also be explicitly applied to a pattern at an inner level since that pattern is always directly accessible through its unique identifier within the *Hierarchical Pattern*.

Increase(n, P, position) Inserts “n” component place-holders into pattern “P” according to its structural definition, but at a specific position within that structure. Namely, that position is related with a specific element within “P” which is identified by the value passed as argument to the parameter “position”. This version of the *Increase* operator is to be used in *particular cases* of: S-PTs (e.g. extended non-topological patterns); *Component-Instantiated Structural Patterns (CISPs)*; and *Pattern Instances (PIs)*. Similarly to the previous version of the *Increase* operator, this version is not recursively applied to the inner elements of “P”, in case this one is a *Hierarchical Pattern*.

Decrease(n, P) Decrements the number of component place-holders (i.e. elements not yet bound to executables) of a pattern “P” by the value passed as argument to the parameter “n”. This value should not be greater than the maximum number of existing component place-holders within the pattern. However, if this is not the case and “P” is a partially instantiated *CISP* or *PI*, i.e. some members are already bound to executables, this version of the *Decrease* operator only eliminates all free component place-holders. Similarly to what was described for the *Increase* operator, the *Decrease* is not recursively applied to inner patterns of a *Hierarchical Pattern*, but can be directly applied to those inner patterns.

Decrease(n, P, position) Deletes “n” pattern elements (either elements already instantiated to executable components or free component place-holders) starting at, and including, the pattern element identified by the argument “position”. Consequently, if it is possible to define an ordering for the operated pattern “P” (e.g. *Pipeline* and *Ring* patterns), the element “position”, and its “n-1” consecutive elements are deleted. However, in case it is not possible to define an ordering within the operated patterns, the *Decrease* is to be called with value “1” for the parameter “n”, meaning that only the pattern element identified by the argument value of “position” is removed. This version of the *Decrease* operator is also not recursively applied to the inner elements of “P”, in case this one is a *Hierarchical Pattern*.

Extend(P) A new member is added to the structure of pattern “P” considering a structural recursive iteration permissible in the context of that pattern’s semantics. The place where that new component place-holder (CPH) is placed is pre-defined and it is dependent on the structural definition of “P”. For example, extending an *Adapter* pattern (introduced in Figure 3.7) implies annotating the new CPH as another *Adapter* for the previously existent *Adapter*, which then becomes the *Adaptee* of that new CPH. In this way, the pre-existent (first) *Adaptee* is adapted twice. This is useful, for example, if the firstly defined *Adapter* element is not fully conform to the requirements of a new configuration.

Extend(element, P) Augments the structure of “P” by one recursive iteration, similarly to the previous version of the *Extend* operator, but allowing the definition of the place for the new CPH. Specifically, the point where the structure of pattern “P” is to be augmented is defined by the value passed as argument to the parameter “element”. As such, “element”

represents a pre-existent member of "P" which defines a structural position where it is meaningful to extend that pattern's structure. For example, in case of a *Facade* pattern that had already been augmented by the *Extend* operator, it is possible to augment again the inner (pre-existent) *Facade* by generating an intermediate *Facade* between the two existent *Facades*.

Reduce(P) Truncates one structural recursive iteration of the pattern "P" in the inverse order that resulted from the application of the *Extend(P)* operator. Hence, this operator is symmetric to the *Extend* operator in the way that one operation of *Reduce(P)* undoes one structural iteration resulting from the *Extend(P)* operator. Taking as example the one presented in the description of the *Extend(P)* operator version, reducing the extended *Adapter* pattern implies eliminating the lastly added *Adapter*.

Reduce(element, P) Truncates one structural recursive iteration of pattern "P" but that is related to the structural position defined by the parameter "element". Consequently, the value passed as argument to parameter "element" has to identify a particular member within the previously extended structure of "P" where it is possible to generate a coherent contract of its the structure. Evoking the example described in the *Extend(element, P)* operator, it is possible to use this extended version of the *Reduce* operator to eliminate, for example, the most inner *Facade* within the previously extended structure.

Group(P1, .., Pn, ResultP) Aggregates a set of Patterns identified by the parameters "P1", "P2", ... , "Pn", forming a *group* (also sometimes referred as an *aggregate*). This newly created Structural Pattern, which is named after the value passed to the parameter *ResultP*, represents its members as a single entity, with no other structural relationship among those members. If the argument to the "ResultP" parameter refers to an already existing group, patterns "P1".."Pn" are added to the group. Moreover, if the arguments "P1".."Pn" are themselves groups, they are merged into a single group identified by "ResultP" (which may exist already).

UnGroup(P) On one hand, if the pattern "P" was the result of the *Group* operator, the aggregation is dissolved and "P" disappears, but the inner patterns continue to exist. Furthermore, if those inner patterns also are groups themselves, they remain intact, i.e. the *Ungroup* operator is not recursive. On the other hand, if the pattern "P" passed as argument to the *Ungroup* is not an aggregate, this operator has no effect. Since the *Ungroup* operator undoes the result of a previous *Group* operation, the former operator is said to be symmetric to the latter operator.

Embed(P1, P2, position) Includes a pattern "P1" into another pattern "P2", specifically in the component place-holder identified by the parameter "position". The concept of hierarchy is therefore supported here by enabling component place holders to contain other PTs.

Embed(P1, P2) This version is used to include pattern "P1" into pattern "P2" when it is not necessary, or possible, to identify a position within "P2", e.g. when "P2" is an aggregate that resulted from the *Group* operation. In case "P1" is itself an aggregate, the operator encloses the group "P1" into group "P2" as its member, i.e. "P1" still exists as a group

within “P2”. In this way, this version of the *Embed* allows the construction of a hierarchy of groups contrarily to the *Group* operator that would perform a merge of the two groups passed as arguments.

Extract(P1, P2, position) Removes pattern “P1” from within pattern “P2”, specifically from the element identified by “position”. Pattern “P1” is moved to the same level as the outmost pattern that may enclose “P2”. This operator is complementary to the *Embed* Operator.

Extract(P1, P2) This version is used to remove pattern “P1” from within pattern “P2” when:
a) it is not possible to identify the particular position where “P1” is located within “P2”, i.e. when “P2” is an aggregate that resulted from the *Group* manipulation; or b) the value of parameter “P1” is sufficient to locate the pattern to be extracted from “P2”.

3.3.3 Inquiry Operators

These operators are used to inquire about the properties of the patterns and perform compatibility checks. Inquiry Operators, in general, return a boolean result and include:

IsExtensible(P) Identifies if pattern “P” has an extensible structural semantics, returning a true or false value. This operator may be used to check if the *Extend/Reduce* Structural Operators can be applied to “P”.

IsHierarchical(P, type) Verifies if pattern “P” is a *Hierarchical Pattern* which may have resulted from the application of either the *Group* or *Embed* operators, returning a true or false value. In case of a successful check, this operator returns a true value and the output parameter “type” defines if “P” is a group pattern (“type” returns the value “G”) or if “P” is hierarchical as a result of an *Embed* manipulation (in this case, “type” returns the value “E”). In case this operator returns a false value, the parameter “type” has no meaning.

IsInHierarchy(P1, P2, position) Verifies if pattern “P1” is one of the elements of the *Hierarchical Pattern* “P2”. The verification is based only on the unique identifier of “P1” within the system, and it is done recursively in all elements within “P2” which are also *Hierarchical Patterns* themselves. In case the pattern “P1” is found, a true value is returned, and the output parameter “position” is instantiated with the path name to access “P1” within “P2” (e.g. “P2.P(i).P1”, where “P(i)” represent the chain of names of the successive intermediate hierarchical patterns that enclose “P1”). In case the pattern “P1” is not found, a false value is returned, and the output parameter “position” has no meaning.

EqualStructure(P1, P2, depth) Verifies if two patterns have exactly the same structure, returning a true or false value. This operator also accepts *Hierarchical Patterns* as arguments performing a recursive checking until the hierarchic level defined by the parameter “depth” is reached. If “depth” is equal to the “zero” value, this operator only checks if the patterns “P1” and “P2” are instances of the same *Structural Pattern* independently of their number of elements. For example, in case “P1” and “P2” are two pipeline S-PTs

with a different number of elements, the *EqualStructure* operator would return true. If “depth” is greater or equal to the value “one”, the operator first performs a check on level “one” evaluating if “P1” and “P2” have the same number of elements and if each element of both patterns is of the same structural type (i.e. if the elements under comparison are two instances of the same *Structural Pattern*). Subsequently, the verification is done recursively until, and including, the hierarchic level defined by the parameter “depth”. The exact semantics of this operator is implementation dependent since it relies on the definition of an ordering within the Structural Patterns (e.g. the “satellites” of a *Star S-PT* would be created in a well defined order allowing the comparison of the “satellites” of two *Star PTs* according to that order).

IsSubstructure(P1, P2, position) Verifies if a pattern “P1” is a sub-structure of the *Hierarchical Pattern* “P2”, returning a true or false value. Pattern “P1” may also be a *Hierarchical Pattern*. The verification checks for the existence of a pattern enclosed in “P2” whose structure is similar to “P1” (analogously to the verification done by the *EqualStructure* Operator but with no depth limit). If a similar enclosed pattern is found, the operator returns a true value and the output parameter “position” returns the path name of that pattern within “P2”. Otherwise, the operator returns a false value and the parameter “position” has no meaning.

IsCompatible(P1, P2) Verifies if a pattern is compatible with another one. This operator is used to determine if two patterns are functionally identical. This analysis is undertaken in stages. The first stage involves checking if two patterns are structurally similar (e.g. analogously to the *EqualStructure* operator). The second stage involves checking if the control and data flows between components within a pattern are similar. The final check involves verifying if all components (or types) within two patterns are identical. All three checks must be valid for the compatibility test to pass, and the exact semantics of the checks is implementation dependent.

IsOwner(P1, {A1, ..., An}) This operator is used to confirm if one particular user or a group of users identified by the parameters “A1”.. “An” are the owners of pattern “P1”. This operator is related with the *Ownership Operators* ahead, and may be used, for example, to check if a subsequent operator manipulation of “P1” is allowed for the user(s) identified by “A1”, ..., “An”.

The presented operators above simply aim to suggest a few possible ways to inquire about patterns’ characteristics and that, therefore, may be somehow useful on supporting the user on the application of the other types of pattern operators upon the inquired patterns. Such support on application configuration is important either if the construction process is based on scripting language or if the user works with a visual environment provided by a *PSE*. Nevertheless, since we consider that a full understanding of the applicability of these *Inquiry Operators* is not crucial to the clarification of the the overall relevance of the pattern operator model, a full description of their semantics is not presented in this work.

3.3.4 Ownership Operators

These operators relate to protection and access rights concerns and assume a definition of user ownership and protection concepts, which are also not developed in this thesis. Therefore, the description below only aims at illustrating how the proposed model can also integrate those kind of operators. Particularly, we present a few operators that may allow defining the users who may modify a pattern in general (i.e. all operations upon that pattern are allowed), or the users who are granted the permission to perform a particular set of operations. For the first case, it is also possible to restrict the users' access rights to a specific time period.

DefineOwners(P1, {U1, ..., Un} [, δT]) This operator is used to make a particular user or the set of users identified by "U1" ... "Un" the *owners* of pattern "P1". Each U_i may also represent a user group and hence user identification is implementation dependent. All the users defined as *owners* have thereafter modification rights to the pattern "P1" but the associated set of allowed operations is also implementation dependent. This might indicate, for example, that all users with a *owner status* for pattern "P1" may manipulate "P1" with any of the defined pattern operators. The δT parameter, in turn, is optional and defines a time interval during which the *owner status* for the defined users is valid. The expiration of that time interval is equivalent to a call to the *UndefineOwners* operator below referring the same users.

UndefineOwners(P1, {U1, ..., Un}) This operator disables the modification rights for the users "U1" ... "Un" which were previously granted by the *DefineOwners* operator. In case the owner status had been granted for a specific time interval and this has already expired, the *UndefineOwners* operator has no effect.

AssignActivity(P1, {Activity}, {U1, ..., Un}) Enables pattern "P1" to be modified by users "U1" ... "Un" according to the set "Activity". The operations in the set "Activity" may either be pattern operator invocations or user defined operations (e.g. which are bound to a particular implementation). The set of users identified by "U1" ... "Un" acquire the *owner status* for pattern "P1" but only to perform the actions represented by the set "Activity".

RemoveActivity(P1, {Activity}, A) Disables a single or a set of activities represented by "Activity" for pattern "P1" and users "U1", ..., "Un".

Please note that the owner of a pattern may delegate its access to a single user or group of users by calling the *DefineOwners* operator upon the same pattern (the same applies to the *AssignActivity* operator).

Ownership Operators, as the ones described above, can hence define ways to control application construction/manipulation which may be significant for many application domains including the Grid (since Grid environments typically provide support to many different types of users).

3.3.5 Execution Operators

Executing a pattern involves the coordinated execution of its components. Specifically, we model the *execution of a pattern instance* as a *distributed computation* as specified by Marzullo and Babaoglu in [217]. Namely, a distributed computation is defined by a partial ordering on the set of events which can be generated by the execution of individual components ("local events") and their interactions ("interaction events"). The partial event ordering is induced by data and control flow and time dependencies among components, as defined by Behavioural Patterns (combined with Structural Patterns), which put synchronisation constraints on the execution ordering.

The *state of an execution of a Pattern Instance (PI)*, on the other hand, is characterised as the global state of the distributed computation associated to that pattern instance. Such global state includes both the data and execution states, as well as the data at the communication links. This is defined using the same concept from distributed computing theory as mentioned above [217].

Therefore, the semantics of the defined *execution operators* rely on the above concepts of "executing a pattern instance" and "the state of a pattern instance". Considering their operational perspective, execution operators are activated through execution scripts acting upon the particular resource management system being used (e.g. Globus [11] for a particular Grid environment).

This sub-section provides a summary description of the available execution operators. The complete description of each operator's semantics is deferred to section 4.4.

Start(P) This operator starts a pattern's execution.

Terminate(P) This is used to terminate a pattern's execution.

Stop(P) This operator is used to pause a pattern's execution – with the side-effect of saving a checkpoint of the execution state.

Resume(P) The pattern's execution is resumed and the saved checkpoint state, which resulted from the application of the *Stop* operator, is restored. Therefore, this operator is companion to the *Stop* operator.

Repeat(n, P) The execution of pattern "P" is repeated "n" times.

TerminateRepeat(P) Discontinues the action triggered by the *Repeat* operator, thus preventing the launching of further iterations, and ceases the current execution in case an iteration had started in the meanwhile.

Limit(δT , P) Limits the duration of the execution of pattern "P" to a maximum time interval equal to δT . If δT expires before the pattern executing being completed, its execution is terminated by force.

UndoLimit(P) Undoes the time limit set by the *Limit* operator upon the pattern "P", meaning that this pattern execution it is no longer interrupted until it is completed.

Restart(δT , P) Repeats the execution every δT time (periodic execution). This operator allows the user to specify periodic re-starts of an application.

TerminateRestart(P) Discontinues the action triggered by the *Restart* operator by preventing the periodic execution of pattern P which had been previously set by the *Restart* operator.

Log(id, δT , P) This operator is used to periodically checkpoint or log the execution state of a pattern "P". The value of the δT parameter is used to make the distinction between the request for a single checkpoint or a periodic log. Namely, if δT is zero, a single checkpoint is accomplished, and it is identified by the parameter "id". However, if δT is greater than zero, this value defines the time interval for the periodic log. In this situation, the parameter "id" is to be associated to a time tag, one for each periodic log, forming another identifier ("idt") that will uniquely identify each specific log operation. Consequently, a set of identifiers (i.e. several "idt" tags) will identify the log operations, each one undertaken at a particular time (which is ruled by the parameter δT). Those identifiers can be used as arguments to *ResumeLog* operator ahead, to restore execution from a particular saved pattern's execution state.

TerminateLog(P) Discontinues the action of the last periodic logging set by the *Log* operator.

SeqLog(P) Triggers an automatic mechanism for collecting a succession of intermediate saved global states of "P" each one triggered with a unique identifier. Such succession constitutes a trace of the pattern's execution and allows its off-line inspection by appropriate tools. Each of the generated identifiers, i.e. one for each saved intermediate global state, can also be passed as argument to the *ResumeLog* operator ahead to repeat the execution of pattern "P" but from that saved state.

TermSeqLog(P) Interrupts the action of the sequential logging set by the *SeqLog* operator.

ResumeLog(idt, P) The execution state of pattern "P" is resumed from one previous logged state that resulted either from the *Log* or *SeqLog* operators, and to which a unique tag, generated in the context of those operators, was associated. Consequently, the parameter "idt" is to be instantiated with a tag generated in the context of the *Log* or *SeqLog* operators which identifies that particular saved state.

SteerComponent({parameters}, P, component) Change the value of a set of parameters of a specific element within pattern "P". The pattern's element is identified by the operator parameter "component" that represents a specific steerable executable/service. Therefore, the modifiable parameters are application dependent.

Steer({parameters}, P) This operator allow changing the value of a set parameters associated to pattern "P" as whole. Those parameters are implementation dependent and may, for example, capture parameters related with the executable/services that instantiate the elements within "P" (e.g. parameters that are common to all executables within "P").

To conclude, these operators have a direct impact on how execution of components within a pattern takes place, and therefore need to interface to existing resource management and scheduling systems. Namely, the operators which refer to a notion of time associated to the

execution of a pattern, rely on the notion of time, as provided by the underlying distributed system. The same applies to the *Global Coordination Operators* presented in the next sub-section. Section 6.5 in Chapter 6 describes, in particular, the mapping of some of the *Execution Operators* described above to the *Distributed Resource Management Application API (DRMAA)* [43].

3.3.6 Global Coordination Operators

This sub-section describes possible operators to define/change the data and control flow semantics, to specify coordination rules associated to patterns, and to manage the connection between patterns.

Similarly to *Ownership* operators (requiring an associated definition of a user/protection/access rights sub-system), the following operators assume an associated definition of a coordination sub-system and policies. The following description in this section only aims at illustrating several possibilities for defining coordination capabilities through specific Behavioural Operators. As such, their semantics was not defined in this work, and the following description only suggests possible additional desired functionalities to manipulate Behavioural Patterns.

DefineRoleBehavPatt(P, B-P, {element, role}) Annotates one or more specific elements within pattern “P” with roles defined by the semantics of Behavioural Pattern “B-P”. The parameter “{element, role}” represents several mappings, one for each “element” within “P” and its correspondent behaviour at execution time according to pattern “B-P”.

DefineBehavPatt(P, B-P) Applies the *Behavioural Pattern* identified by “B-P” to the entire pattern identified by “P”, i.e. all elements within “P” are annotated with specific roles within “B-P”. The mappings between the elements of “P” and the “roles” within “B-P” are pre-defined and are implementation dependent. This operator hence avoids the need to explicitly define individual behaviours to all elements of “P”.

ReplaceBehavPatt(P, B-P1, B-P2) Replaces the Behavioural Pattern “B-P1” as a single entity, with the Behavioural Pattern “B-P2”, in the context of a configuration named “P”. Concretely, “P” represents a specific Structural Pattern combined with the Behavioural Pattern “B-P1” (e.g. a *SB-PT*), and after the behavioural replacement, “P” corresponds to the same Structural Pattern now combined with the Behavioural Pattern “B-P2”. Similarly to the *DefineBehavPatt* operator, the mappings between the elements of “P” and roles within “B-P2” that coordinate their behaviour are pre-defined and are implementation dependent.

Coordinate(P, rule) Apply the coordination rule to elements of pattern “P”. The rule is identified by the parameter “rule” and defines, for example, the temporal/data flow/control flow dependencies between the elements. Consequently, the rule may be constructed as a sequence of *Execution* and other Behavioural Operators (e.g. the rule may represent the sequence “*Stop, ReplaceBehavPatt, Resume*”), or it may represent a set of coordination rules supported by the implementation system (e.g. each rule can be defined using the *deftemplate-defrule* structure found in the Java Expert System Shell (JESS) [29]).

ChangePatternDependencies(rule, P) This operator allows the execution environment or a user to change the dependencies between elements of pattern “P” according to the “rule” parameter. The rule may change, for example, the following type of dependencies:

- control flow dependencies, e.g. the control flow within a pipeline PI is changed from a push-based control rule, to a pull-based control rule;
- data flow dependencies, e.g. the data flow in a pipeline is reversed.

Similarly to the *Coordinate* operator, the “rule” parameter may represent a set of other Behavioural Operators, or a set of coordination rules supported by the implementation system.

DefineDependencies(rule, P1, ..., Pn) This operator allows the execution environment or a user to define/change the interdependencies between the set of (unrelated) patterns “P1”, ..., “Pn”. The rule may change independently for the following type of dependencies:

- time dependencies, e.g. all patterns have to produce their results in a synchronous fashion;
- control flow dependencies, e.g. all patterns are executed in a round-robin fashion, as soon as one terminates execution, the next one begins;
- shared data dependencies, e.g. change the way the set of patterns access a shared resource (e.g. to switch from exclusive access to permission to multiple entities).

The “rule” parameter may again represent a set of coordination rules supported by the implementation system.

3.3.7 Pattern and Operator Summary

Table 3.1 summarises the list of patterns and operators. Specifically, the table shows the operators to manage the defined/selected Structural Patterns, and the Behavioural Operators that can manipulate Structural Patterns combined with Behavioural Patterns.

3.4 Summary

This chapter described the main characteristics of a pattern- and pattern operator-based model for supporting a structured development and execution control of applications over Grid environments. A set of Structural and Behavioural Patterns were also discussed in this Chapter, whereas the discussion on the semantics of the Structural and Behavioural Operators is presented in Chapter 4. The reconfiguration strategies possible in the context of the model are also deferred to Chapter 5.

The model also supports a methodology for application construction which may support the systematisation of some of the necessary steps for application development. We claim this is specially useful for supporting the building and configuration of Problem Solving Environments, as illustrated in the examples of Chapter 7.

	Patterns	Operators
Structural	Pipeline, Star, Ring, Adapter, Proxy, Facade	Create, Eliminate, Replicate, Reshape, Replace, Increase, Decrease, Extend, Reduce, Embed, Extract, Group, Ungroup, IsExtensible, IsHierarchical, IsInHierarchy, EqualStructure, IsSubstructure
Behavioural	Master-Slave, Streaming, Client-Server, Peer-to-Peer, Mobile Agents/Itinerary, Remote Evaluation, Code-on-Demand, Contract, Observer/Subscribe-Publish, Parameter Sweep, Service Adapter Pattern	IsCompatible, IsOwner, DefineOwners, UndefineOwners, AssignActivity, RemoveActivity, Start, Terminate, Stop, Resume, Limit, UndoLimit, Repeat, TerminateRepeat, Restart, TerminateRestart, Log, TerminateLog, SeqLog, TermSeqLog, Steer, SteerComponent, DefineRoleBehavPatt, DefineBehavPatt, ReplaceBehavPatt, Coordinate, ChangePatternDependencies, DefineDependencies

Table 3.1: *Pattern Templates and Operator Summary.*

4

Pattern Operator Semantics

Contents

4.1	Introduction	84
4.2	Semantics of Structural Operators	84
4.3	Sequences of Structural Operators	106
4.4	Semantics of Behavioural Operators	115
4.5	Sequences of Behavioural Operators	129
4.6	Summary	133

This chapter describes the semantics of the Structural and Behavioural Operators.

4.1 Introduction

This chapter describes the semantics of the operators introduced in Chapter 3. Namely, the chapter begins with the Structural Operators, followed by the description of some Structural Operators sequences. Subsequently, the chapter describes the semantics of the Behavioural Operators and the description of applying some of these operators in sequence. The discussion on sequences including both Structural and Behavioural operators are deferred to Chapter 5.

4.2 Semantics of Structural Operators

Structural Operators	Topological Patterns (Pipeline, Star, Ring)	Non-Topological Patterns (Adapter, Proxy, Facade)
Replicate, Replace, Embed, Extract, Group, Ungroup	Applicable to all	Applicable to all
Increase, Decrease	Applicable to all	Non-applicable to the Adapter pattern
Extend, Reduce	Non-applicable	Applicable to all
Reshape: – to restructure a pattern into a topological pattern	Applicable to all	Applicable to all
– to restructure a pattern into a non-topological pattern	Depends on the cardinality of the pattern templates	Depends on the cardinality of the pattern templates

Table 4.1: *Applicability of Structural Operators to Topological and Non-topological Structural Pattern Templates*

Structural Operators generate or manipulate *Structural Patterns* guaranteeing that the structural constraints of the created or manipulated patterns are preserved in the process. This means that the resulting instances remain consistent with the semantics of the Structural Patterns they represent. Nevertheless, not all Structural Operators are applicable to all Structural Patterns, as represented in Table 4.1. Moreover, the produced result of some operators (e.g. *Reshape*) is dependent on the patterns they are applied to. Such restrictions will be explained in the discussion of each operator’s semantics.

One should also note that additional transformations to the operated Structural Patterns may always be undertaken by a user directly using an editor – although this does not necessarily provide any checking that the transformation will leave a pattern class invariant. Albeit such consistency checking is desirable as future work, the work presented in this thesis only considers application configuration and execution control in a pattern-based perspective.

The next two-subsections describe the Structuring and Grouping operators, but only when applied to (*Structural Pattern Templates* (*S-PTs*)). The manipulation of *SB-PTs*, *CISPs*, and *PIs* by those operators is deferred to 5.2.

4.2.1 Structuring Operators

The two basic *Structuring Operators* are the *Create* and *Eliminate* operators for generating a particular *Structural Pattern Template* (S-PT) and deleting it, respectively.

The *Create*(*SP*, *name* [, *nElems*]) operator generates an instance of the type of Structural Pattern defined by the parameter “SP” according to its structural semantics. The identification and number of elements of the created S-PT are defined by the second and third parameters, namely “name” and “nElems”, respectively. The parameter “nElems” in the *Create* operator is defined as optional since some Structural Patterns may have a fixed number of elements or in case the implementation supporting system creates a S-PT with a default number of elements. The generated component place-holders (CPHs) for the S-PT are to be labeled with unique and meaningful identifiers within that S-PT (called its *CPH_identifier*)¹. To manipulate the S-PT, the access to one of those CPHs is made by composing its name with its S-PT’s name, in the form: “S-PT_identifier.CPH_identifier”.

As an example, Figure 4.1 represents the generation of two S-PTs with a specific number of elements: a Star Structural Pattern (SP) named “starPT” with four component place-holders, namely, a “nucleus”, and three satellites – from “satellite1” to “satellite3”; and a Facade SP named “facadePT” whose interface is represented by the “facade” component place-holder, and the component place-holders representing the Facade’s sub-systems are named “subsyst1”, “subsyst2”, etc. In this example, the *cardinality* of the S-PT “starPT” is equal to four elements, and the cardinality of the S-PT “facadePT” is equal to five.

In turn, the creation of the Adapter S-PT represented in Figure 4.1, i.e. “adapterPT”, does not require the definition of the number of component place-holders to be generated since the *Adapter* Structural Pattern is defined as having two fixed elements for a non-extended Adapter. Moreover, the parameter “nElems” in the *Create* operator may also be ignored in case the implementation support generates a default number of component place-holders for any kind of S-PT.

Implementation-wise, it is also possible to define an order for the component place-holders (CPHs) within a specific S-PT, and to use it to identify a specific CPH within that S-PT. For example: the CPH representing the first stage of a Pipeline S-PT may be the “first” element, the second-stage CPH may be the “second”, etc; the “facade” element within a Facade S-PT may be the “first”, whereas the first sub-system to be created will be the “second”, and so forth. That ordering value could therefore be used to identify a CPH instead of its name (for example, in application configuration through scripts). Nevertheless, the examples presented in this work rely on a CPH’s name to identify it.

The deletion of a S-PT, may be accomplished by the *Eliminate*(*P*) operator, which removes the S-PT as well as its identifier. For example, the operation *Eliminate*(*facadePT*) deletes the S-PT “facadePT”, and this identifier may be reused.

¹For clarification of the examples given throughout this work, the CPHs’ identifiers try to hint the task of each component place-holder within the semantics of the Structural Pattern they belong to, as it will be presented in the examples in Figure 4.1 ahead. However, sometimes those component place-holders are simply labelled as “cph1”, “cph2”, etc., or their name is simply omitted, when their representation within that structural semantics is somehow obvious, and not necessary for the subject under discussion.

Create(SP, name [, nElems]) ➔

Result S-PT

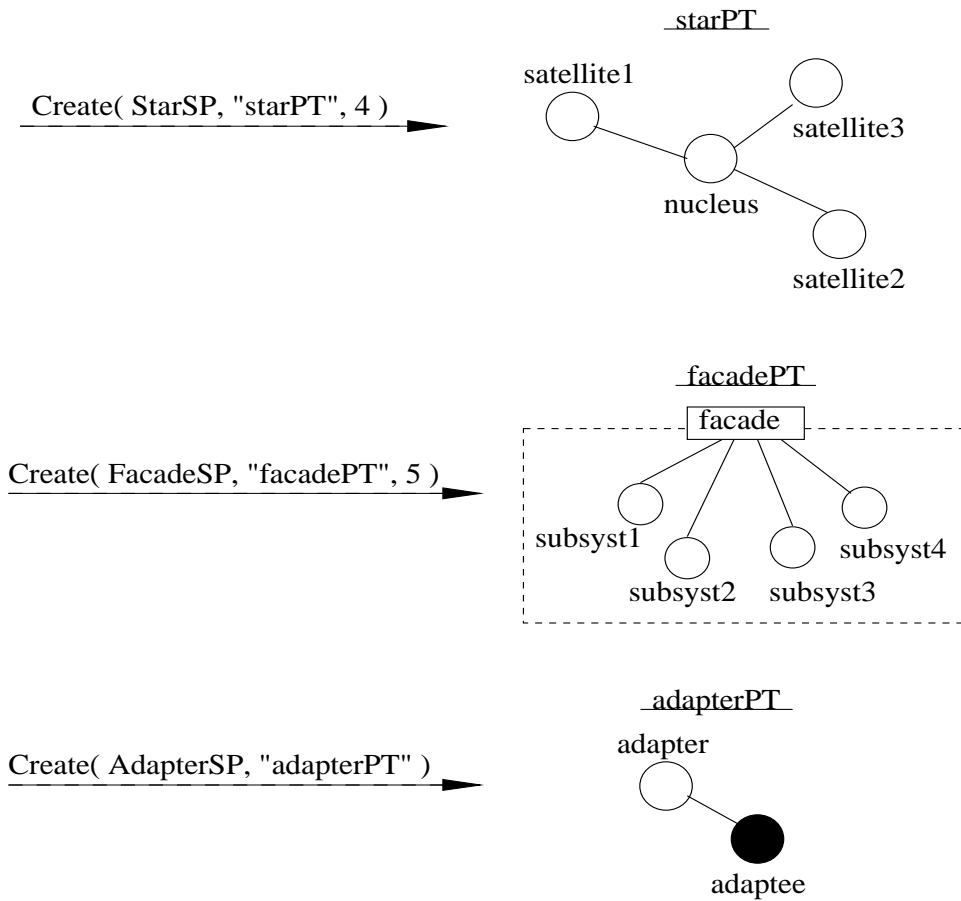


Figure 4.1: The creation of three S-PTs, namely a Star ("starPT"), a Facade ("facadePT"), and an Adapter ("adapterPT")

Replicate, Replace, and Reshape

Both the *Replicate* and *Replace* are operators which do not imply major structural transformations nor are bound to restrictions – all patterns can be replicated and any pattern can be replaced by another one. The *Replicate*(n, P [, { $id1, \dots, idn$ }]) operator creates " n " replicas of the pattern " P " and each replica will have a different identifier (either implementation generated or defined by the parameters " $id1$ ".. " idn "), and the identifiers themselves can be changed. The *Replicate* operator helps the user on creating similar configurations without having to build them from scratch.

The *Replace*($P1, P2$) operator, in turn, substitutes pattern " $P1$ " for pattern " $P2$ " – " $P2$ " takes the place of " $P1$ " within the configuration that included " $P1$ ". This operator allows the reconfiguration of parts of an existing schema, by complementing them with new patterns.

The *Reshape*($P1, P2$) operator transforms one pattern into another, and the cardinality of the pattern being transformed (" $P1$ ") may be important in determining whether the operator can or cannot be applied (see table 4.1). For instance, any topological pattern may be transformed into any other topological pattern, independently of the cardinality of the pattern. For example, a *Pipeline* pattern can be transformed into a *Ring* pattern, by connecting the first and last components of the *Pipeline* (see Figure 4.2). Similarly, a *Pipeline* can be transformed into a

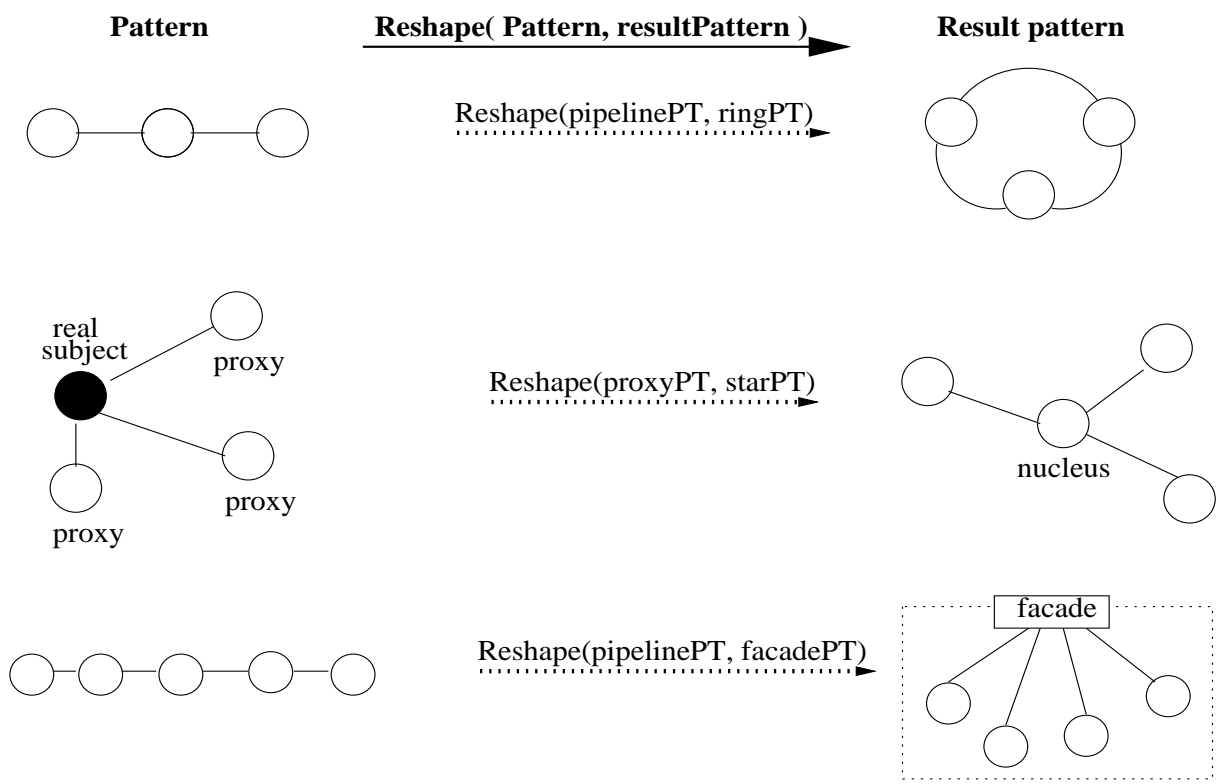


Figure 4.2: Examples of the Reshape operator over a Pipeline and a Proxy Pattern Templates.

Star, by taking one of the *Pipeline*'s components as the *nucleus* of the *Star* pattern, with the other components becoming *satellites*.

Moreover, any non-topological pattern can be restructured into a topological pattern, as long as the cardinality of the original pattern is maintained. For example, in Figure 4.2, a Proxy pattern template containing three proxies is restructured into a star, which will have three satellites. Also, when using the *Reshape* operator, the defined cardinality must be preserved. For example, it is not possible to reshape a *pipeline* pattern with five elements into an *Adapter* pattern, because the latter has two fixed elements. However, the same pipeline can be reshaped into a *Facade* by annotating one of its elements as the *facade* component place-holder, and the other elements as the *sub-system classes* (see Figure 4.2), as a *Facade* pattern is not defined with a fixed cardinality.

The description of the remaining Structuring Operators, namely *Increase/Decrease* and *Extend/Reduce*, is preceded by a functional comparison between those two types of operators.

Comment about Increase/Decrease vs. Extend/Reduce

The pair of *Increase/Decrease* operators manage the incremental growing/decreasing of the structure of a pattern. These operators act at the topmost or higher level (*level one*) of the structural definition of a Pattern. The effect is just to add/delete elements, assuming these elements are of the same type as existing elements in the Pattern definition. This does not imply that all pre-existing elements are of the same type (e.g. a *Star* as a distinguishable element, namely the "nucleus", whereas all the other elements are "satellites"), but that there is a subset of elements in the Pattern definition that is increased or decreased. The semantics of *Increase/Decrease* is

defined according to each type of pattern.

The pair of *Extend/Reduce* operators manage Patterns through recursive extension or reduction based on the Patterns' semantics. Consequently, these pair of operators are not meaningful for Patterns without a clear recursive structural definition or when it is not different from an incremental growing/decreasing of the number of elements (which is already provided by the *Increase/Decrease* operators). Consequently, the *Extend* and *Reduce* operators can be applied to the selected non-topological patterns (i.e. *Proxy*, *Adapter*, and *Facade*), but not to the topological patterns (i.e. *Pipeline*, *Ring*, and *Star*).

Increase and Decrease Operators

As described in section 3.3.2, there are two application versions for the *Increase/Decrease* operators: one that creates/eliminates elements within a pattern' structure in a pre-defined way; and another which requires the user to identify a position, i.e. a particular element within the structure, where to include the new elements or that defines which elements to delete. Since the operated patterns in this discussion are *Structural Pattern Templates (S-PTs)* which include (mainly) undifferentiated *component place-holders (CPHs)*, it is not mandatory to identify a particular element within the structure. Therefore, this discussion concerns mainly the first versions of both operators, whereas a more detailed discussion on the second versions is deferred to 4.3 and 5.2.

The *Increase(n, P)* operator augments the number of elements in a "P" pattern's structure by "n", whereas the *Decrease(n, P)* operators reduce that number by "n". Both operators act according to the structural constraints of the pattern they manipulate. For example, when applied to the *Pipeline* and *Ring* patterns, the *Increase* operator adds "n" stages to those patterns. Figure 4.3 depicts a two-element pipeline PT being increased to a four-element pipeline PT, and Figure 4.4 represents the decreasing of the latter pipeline PT. Being applied to the *Star* pattern, the *Increase* operator increases the number of satellites in the structure, and the *Decrease* operator reduces that number.

Concerning the non-topological patterns, both the *Increase* and the *Decrease* operators may be applied to the *Proxy* pattern resulting in the increasing/decreasing, respectively, of the proxy elements in the pattern, as represented in Figures 4.3 and 4.4. The same operations over the *Facade* pattern, in turn, result in the increasing/decreasing of the number of *subsystem classes* identified as "cph1", "cph2", etc., as shown in the same Figures.

Due to the same reason as explained for the *Reshape* operator, it is not possible to apply the *Increase* and *Decrease* operators to the *Adapter* pattern.

Application-wise, the *Increase* and *Decrease* operators allow, for example: changing the number of stages in sequential data processing applications (which rely on the pipeline-based or ring-based configurations); adjusting the number of processing slaves in a computationally intensive application configured as a star; adding more computational facilities (e.g. more powerful ones) hidden by the same interface provided by a *Facade* (this *Facade* may hence redirect more demanding requests to the new facilities); or controlling the number of local proxies to a remote Web Service.

As for the second versions of the operators, namely *Increase(n, P, position)* and *Decrease(n, P, position)*, Figure 4.5 presents a case of their application to a four-stage pipeline, where

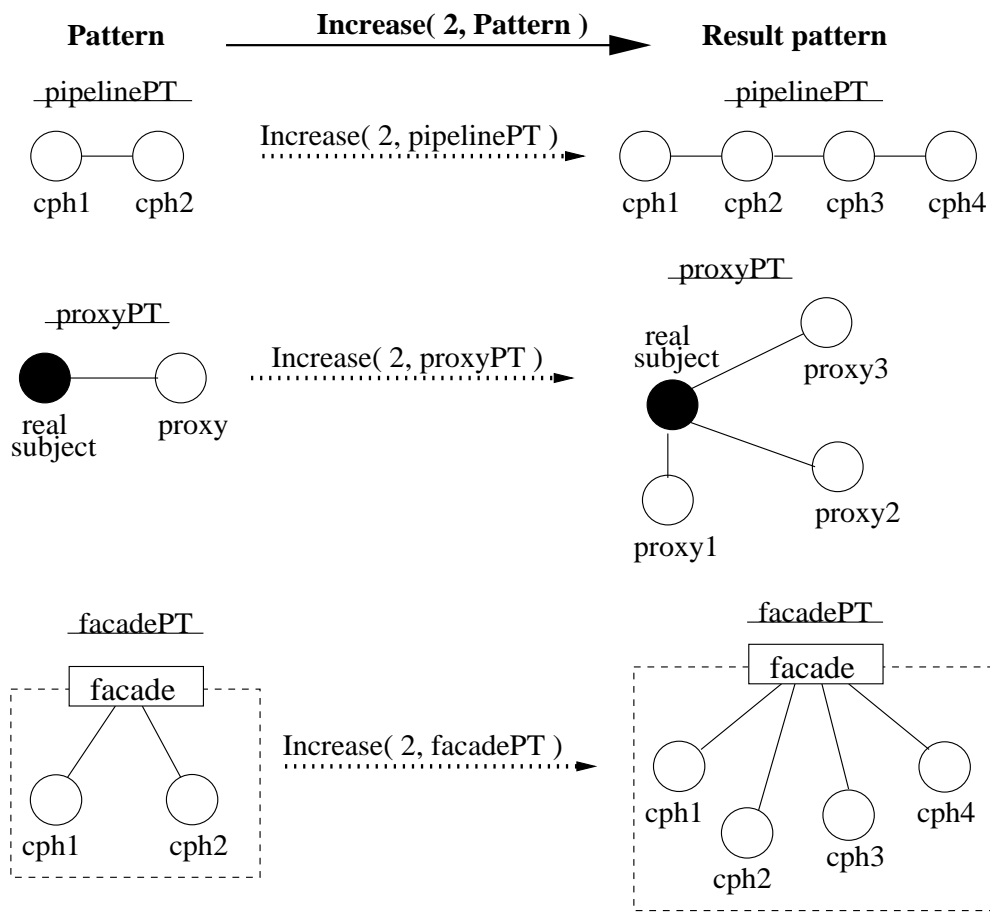


Figure 4.3: Examples of the Increase operator applied to Pipeline, Proxy, and Facade Pattern S-PTs.

the “position” parameter is the component place-holder named “cph1”. On top of the Figure, the application of *Increase(2, pipelinePT, “cph1”)* results on the creation of two new component place-holders (“cph3” and “cph4”) following the “cph1” element of the “pipelinePT” S-PT. As for the *Decrease(2, pipelinePT, “cph1”)* operation, the result is the deletion of two component place-holders, starting at, and including, the “cph1” element. These second versions of the *Increase* and *Decrease* operators are more useful when applied a) to patterns with instantiated elements (therefore differentiated), as will be explained in section 5.2.3; or b) when the structure of a non-topological pattern has been previously extended, as will be exemplified in section 4.3.

Extend Operator

The *Extend* operator is used to augment the structure of a pattern which comprises a recursive definition, where the effect of the operator depends on that pattern’s semantics. There are two possible application versions of the *Extend* operator, namely:

Extend(P) Extends the structure of pattern “P” based on its recursive definition, and both the structural role and the position of the new component place-holder (CPH) are pre-defined.

Extend(element, P) Extends the structure of pattern “P” based on its recursive definition, but

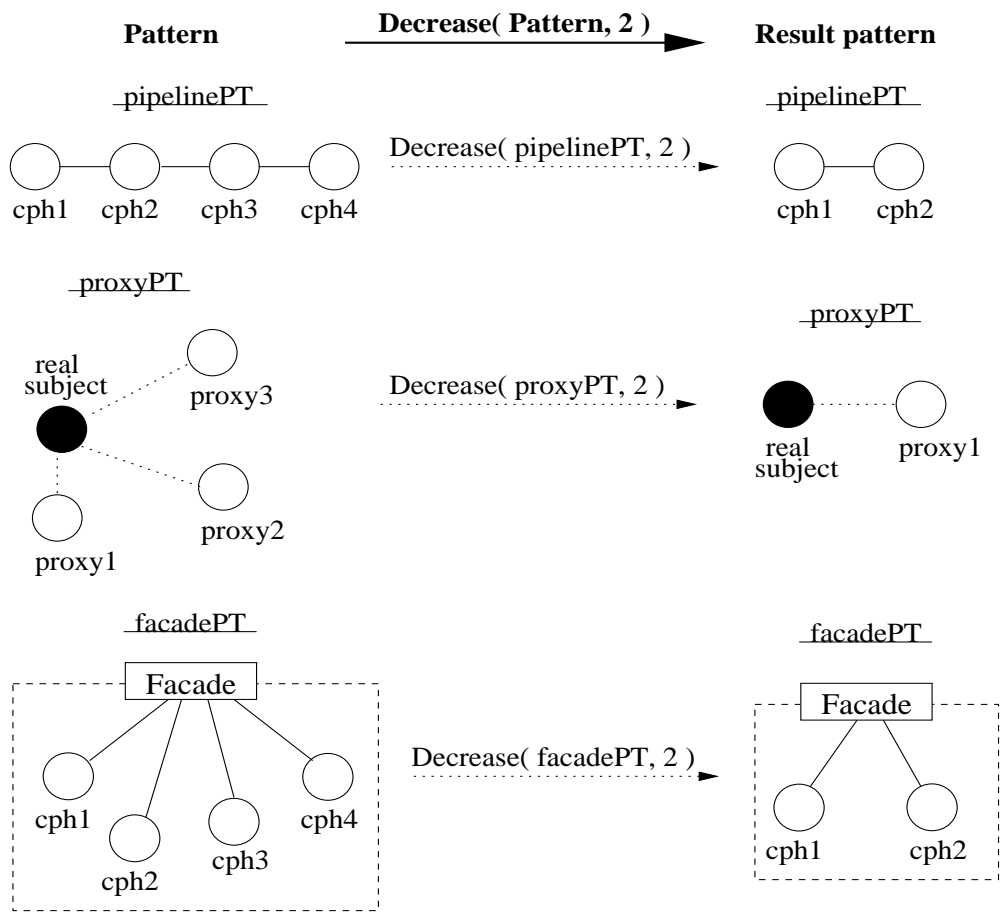


Figure 4.4: Examples of the Decrease operator applied to Pipeline, Proxy, and Facade Pattern S-PTs.

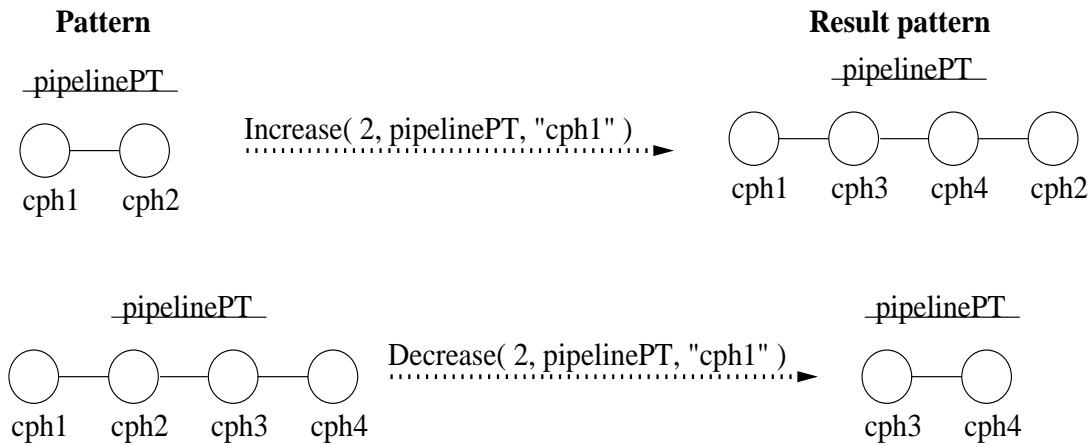


Figure 4.5: Examples of the second application versions of the Increase and Decrease operators upon a Pipeline S-PT.

whereas the structural role of the new CPH is pre-defined, its position depends on the parameter “element”. Concretely, this parameter represents a pre-existent member of “P” and defines a structural position (i.e. element) where it is meaningful to extend that pattern’s structure and hence create the new CPH.

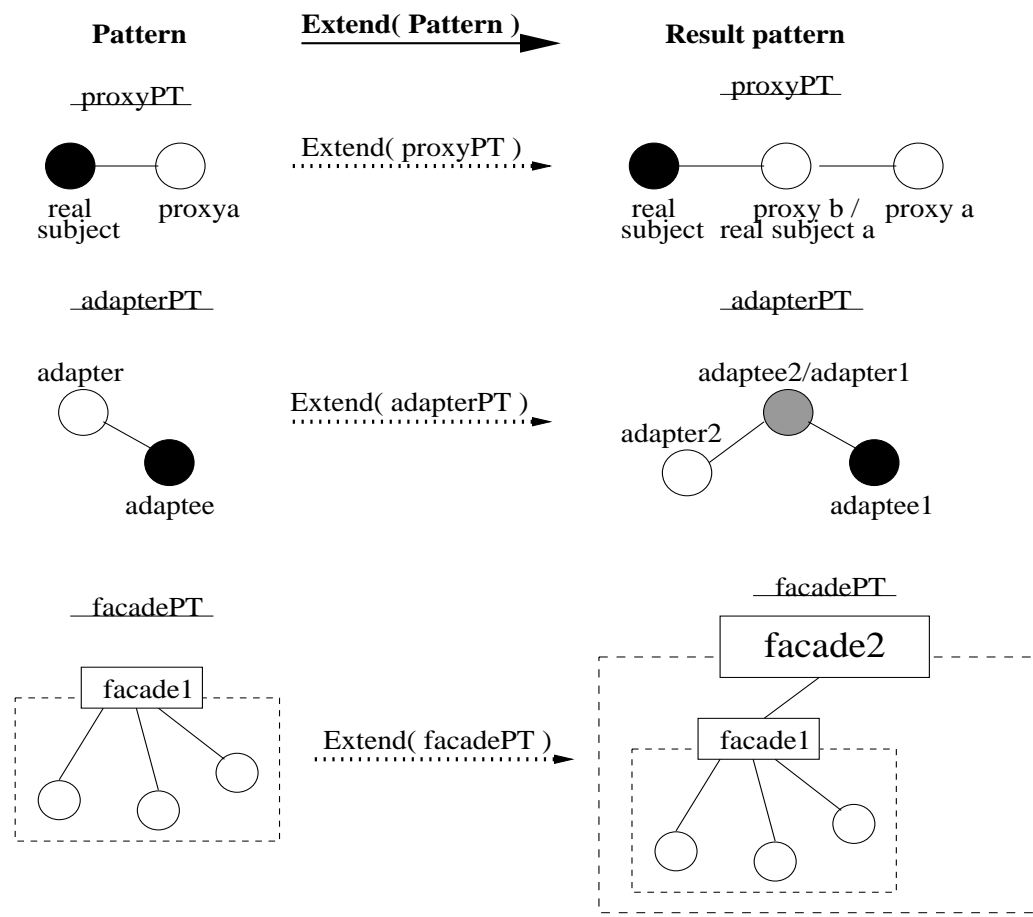


Figure 4.6: Examples of the “ $Extend(P)$ ” operator over cases of the Proxy, Adapter, and Facade Pattern Templates.

Figure 4.6 presents three examples of the application of the *Extend* operator, i.e. $Extend(P)$, to three cases of the *Proxy*, *Adapter*, and *Facade* S-PTs. According to that definition, the location and structural role of the added element is pre-defined and consistent with the semantics of the manipulated pattern.

On top of the Figure, a *Proxy* S-PT is augmented in its structure, and a new CPH with a dual structural role is created as a result. Specifically, the element “proxy b” is created between the elements “real subject” and “proxy a”. Since “proxy b” is defined to represent the “real subject” to the pre-existent “proxy a”, it is also defined as the “real subject a” for “proxy a”. The cardinality of “proxyPT” after the *Extend* operation is equal to three. The extension of a *Proxy* pattern occurs, for example, in mobile agent/object systems, where the sequence of proxies is used for locating the agent/object (via a chain for message forwarders, for instance). Therefore, the *Extend* operation allows the inclusion of an extra proxy that, for example, represents the agent/object or service in case these move to another location.

Likewise, the application of *Extend* to the Adapter pattern (Figure 4.6) would allow a further adaptation of legacy code (reusing the previous adaptation) to allow it to be accessed by other computational components with different interaction requirements. As presented in the middle of Figure 4.6, the operator $Extend(adapterPT)$ creates another “adapter” element within the “adapterPT” S-PT, i.e. “adapter2”, and the pre-existent adapter becomes the “adaptee” element within the structure (i.e. “adaptee2”) but remaining the adapter (i.e. “adapter1”) for

the principal element to be adapted (i.e. “adaptee1”). The cardinality of “adapterPT” becomes equal to three after the *Extend* operation.

Finally, the *Facade* pattern may be extended as presented on the bottom of in Figure 4.6. A new *facade* component within the structure, i.e. “facade2”, “hides” an existing *facade* (i.e. “facade1” on the left-side of the Figure) that becomes a simple *subsystem class* for the new *facade* component. The cardinality of the “facadePT” S-PT increases from the value four to five after the *Extend* manipulation. Such operation may be useful, for instance, for extending the access interface to a set of Grid services in a portal. The first Facade (i.e. “facade1”) might provide an uniform interface to a set of services, and the extension to a new Facade (i.e. the addition of “facade2”) would allow a more complete interface for redirecting services both to: a) new functionalities provided by new (incrementally added) sub-systems; and b) to the pre-existent services.

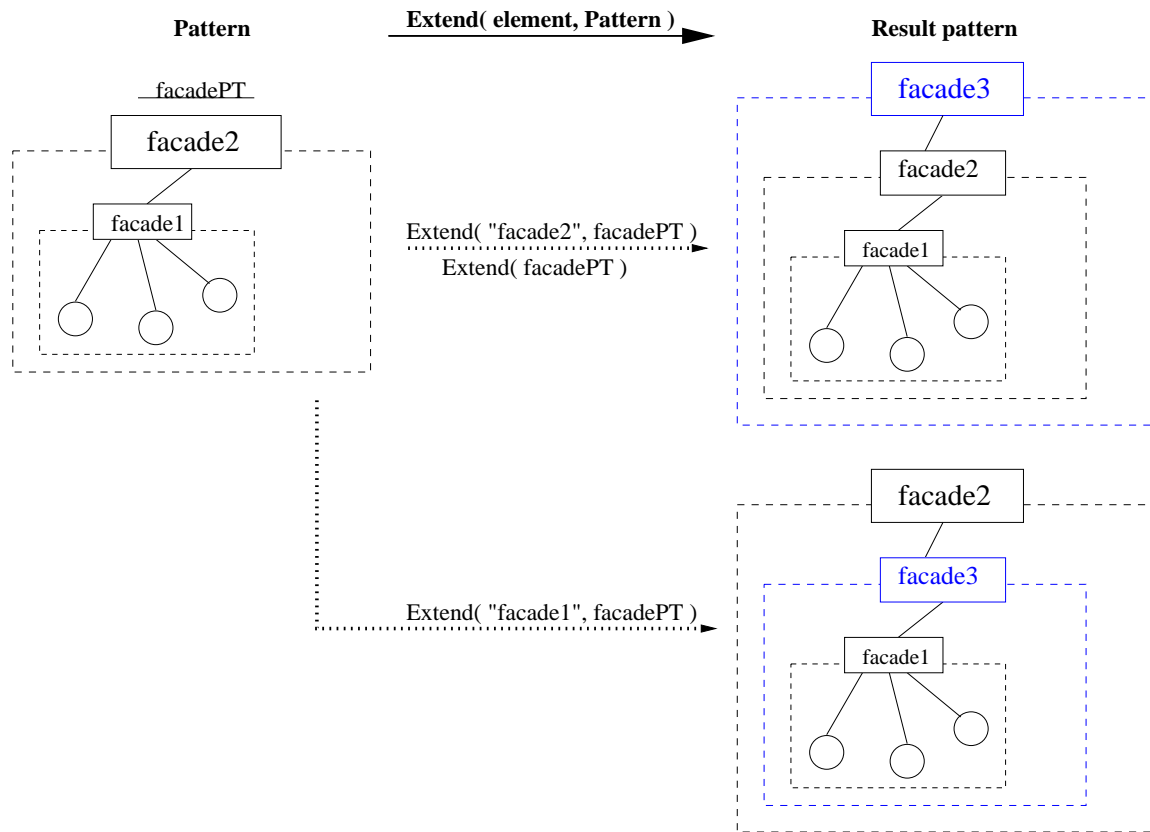


Figure 4.7: Example of the “*Extend(element, P)*” operator over one Facade Template.

To clarify the applicability of the second version of the *Extend* operator, i.e. *Extend(element, P)*, Figure 4.7 presents its use onto a *Facade* pattern. In this second version, the “element” parameter defines a member of pattern “P” where it is possible to apply a recursive iteration on the pattern’s definition. First, and as presented on top of the Figure, the result of applying the *Extend(“facade2”, proxyPT)* operator is similar to the result of applying the *Extend(proxyPT)* operation described before. Particularly, another Facade structure is created – represented by the “facade3” CPH – which in turn interfaces the previous Facade that is represented by the “facade2” element. New sub-systems may be added to the outmost Facade structure, as to any inner Facade. This will be clarified in section 4.3 that discusses interleaved sequential

application of the *Extend* and *Increase* operators.

However, if this extended version of the *Extend* operator is applied to a “facade” structural element within the previously extended “facadePT”, the produced result is different from the one resulting from the first version of the operator. As presented on the bottom of right-side of Figure 4.7, the *Extend*(“facade1”, facadePT) operator generates a third Facade structure, i.e. “facade3”, between the Facades represented by the “facade2” and “facade1” CPHs. Recalling the example of the portal interfacing the access to a set Grid service presented for the first version of the *Extend* operator, the creation of an intermediate Facade structure, might allow, for example, the creation of an intermediate Grid sub-domain when the portal’s structural definition is already set. In this way, the outer Facade represented by “facade2” would not be changed, but the access to the inner facade symbolized by “facade1”, would now be interfaced by the intermediate Facade, i.e. “facade3”. This Facade might represent a Grid (sub-)domain including a different department at the same University which could also contribute with additional (although similar) Grid resources besides the ones already interfaced by “facade1”. Please note that the *Extend*(element, P) operator can only receive as argument for the “element” parameter members of the structure of “P” which are annotated as being of the “facade” type within the structure.

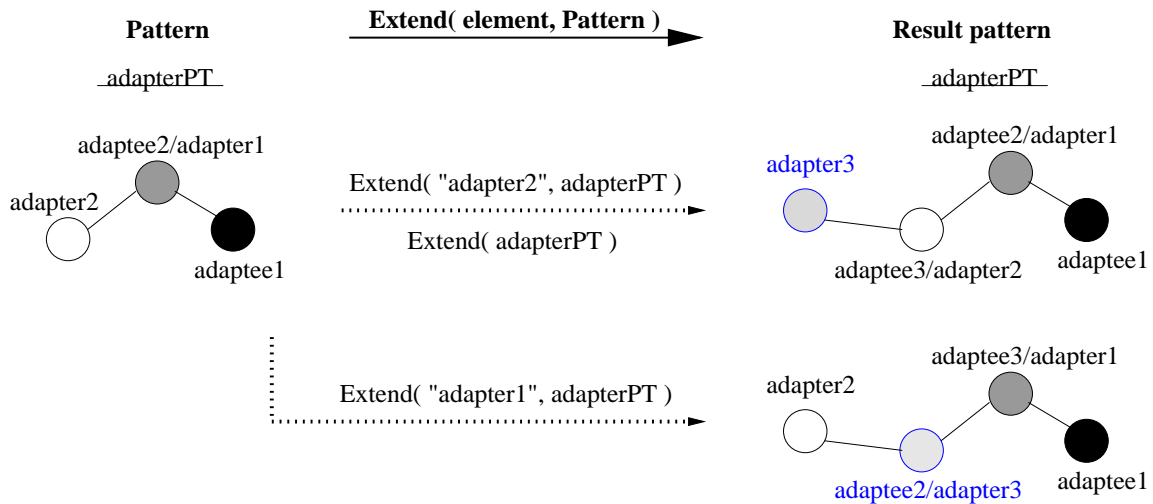


Figure 4.8: Example of the “*Extend*(element, P)” operator over one case of Adapter Template.

As for the applicability of the *Extend*(element, P) to an Adapter S-PT, Figure 4.8 presents two outcomes of that usage. If applied to the “adapter2” element within “adapterPT”, an extra adapter element, i.e. “adapter3”, is created adjusting “adapter2” to be included in a different environment. However, if applied to the inner adapter element, i.e. “adaptee2/adapter1” in the Figure, another intermediate adapter is created, i.e. “adaptee2/adapter3” that now bridges “adapter2” and “adapter1”. The extension of an Adapter structure therefore allows providing a slightly different access to the adaptee element (i.e. “adaptee1”) even when a set of adaptation layers is already defined.

Finally, Figure 4.9 presents the application of *Extend*(element, P) to a Proxy S-PT. On top of the Figure, the result of applying the *Extend*(“real subject”, proxyPT) operator is similar to the result of applying the *Extend*(proxyPT) operation. This means that if the parameter “element”

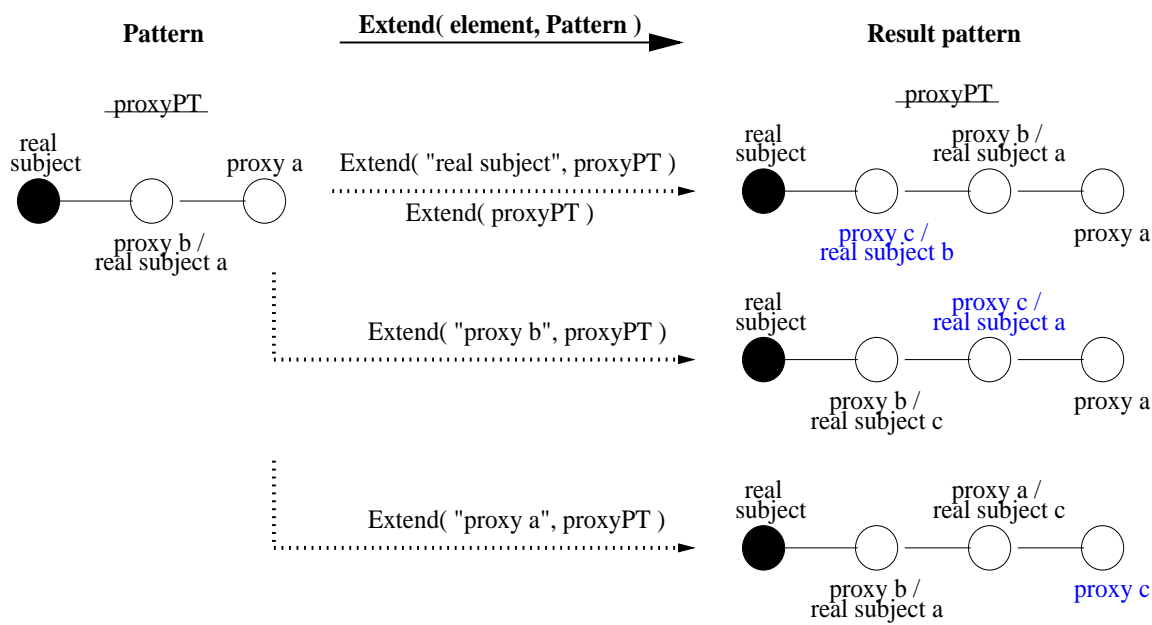


Figure 4.9: Different ways of applying the “Extend(element, P)” operator to a Proxy Template.

is the “real subject”, the *Proxy* S-PT is augmented in the pre-defined way with the creation of “proxy c”, which is similar to what was described before.

In the example presented in the middle of the Figure, the *Extend(“proxy b”, proxyPT)* operation generates a new proxy element, i.e. “proxy c/real subject a” that acts as a surrogate between the pre-existent “proxy b” and “proxy a” elements. This result is as if the “proxy b” has moved to another location, and a new proxy (“proxy c”) is created at the original location of “proxy b”. Another example may be, for example, in case, for some reason, it is not possible anymore to keep the connection between “proxy a” and “proxy b” to access a mobile agent/object (i.e. the “real subject”). This might happen, for instance, in case “proxy a” is to be included in a higher security domain whose access to the outside is now bridged by a new gateway represented by the new “proxy c”.

In the third case (on bottom of Figure 4.9), the parameter “element” in the call to *Extend(element, P)* (second version of the operator) may also be instantiated with the “proxy a” element. The result is that “proxy a” gets its own proxy, namely “proxy c”, and it is now also annotated as “real subject c” acting, therefore, as a surrogate of the “real subject” towards “proxy c”. As it will be exemplified in section 4.3, it is possible to use the *Increase* operator to add new proxies either to the “real subject” element within a *Proxy* pattern, or to members with a double annotation, i.e. a proxy which is also annotated as a “real subject” element like “proxy b/ real subject a” on the left-side of top of Figure 4.9. However, this third application case, i.e. *Extend(“proxy a”, proxyPT)*, represents the transformation of an ordinary proxy into a proxy with a dual annotation (i.e. with both “proxy” and “real subject” definitions), which means that other new ordinary proxies may be directly associated with it. In fact, the practical result is that the chain of proxies may grow towards the right-hand side in the Figure, i.e. new proxies may be added to the first proxy in the chain (i.e. “proxy a”). Such situation might be useful, for instance, in cases when it is not possible to connect a new ordinary proxy with neither the “real subject” nor proxies with a dual definition within a *Proxy* pattern. For example, citing again the case

of a security domain, if this domain already has a proxy to an external service through which all contacts with the service have to be redirected to, it is now possible to create new ordinary proxies within that domain, but that will connect directly the pre-existing internal proxy.

To conclude, we highlight that most of the examples presented in this work make use of the *Extend* operator in its pre-defined form, i.e. *Extend(P)*.

Reduce Operator

The *Reduce* operator is used to lessen the structure of a pattern that was previously subject to an augment by the *Extend* operator. Analogously to the *Extend*, the effect of the *Reduce* operator is also dependent on the semantics of the Structural Pattern it is applied too. Furthermore, there are also two possible application versions of the *Reduce* operator, namely:

Reduce(P) Reduces the structure of pattern “P” by undoing the last recursive structural iteration that resulted from the application of the *Extend(P)* operator.

Reduce(element, P) Reduces the structure of pattern “P” based on its recursive definition, but at the position defined by the argument to the parameter “element”. As such, the “element” must identify a member within the structure where it is coherent to perform a deflation of the previously extended structure of “P”.

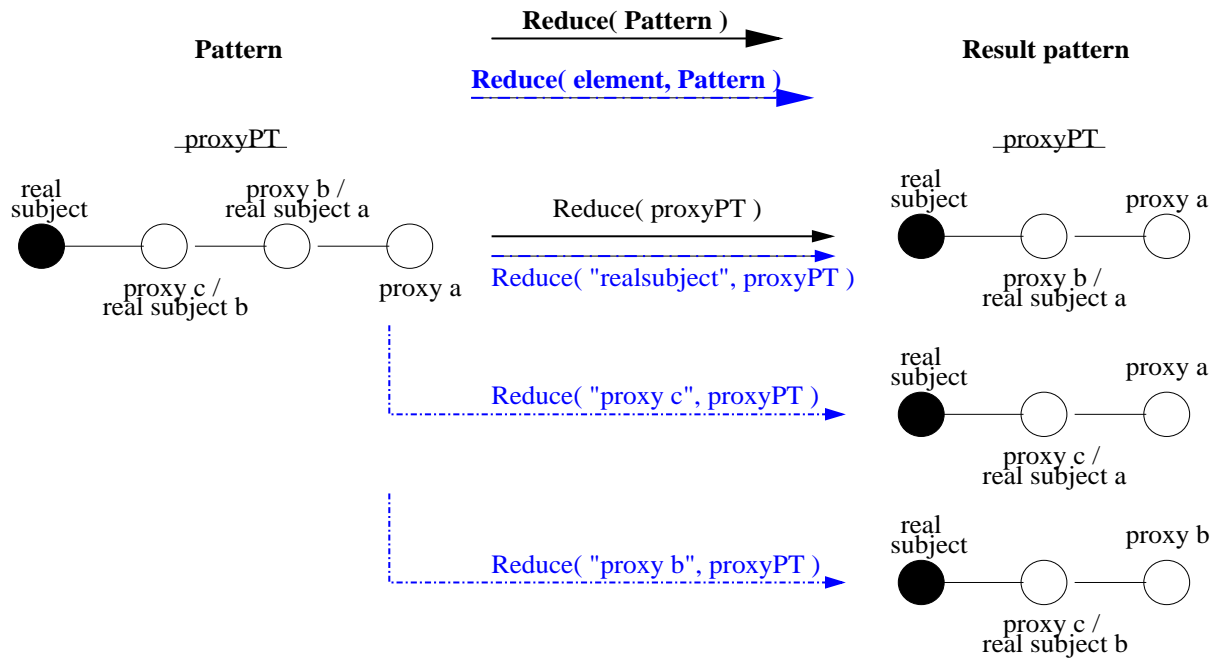


Figure 4.10: Examples of the application of both versions of the *Reduce* operator to one Proxy Template.

Figure 4.10 presents examples of the application of the two versions of the *Reduce* operator to one Proxy S-PT. On top of the Figure, the *Reduce(proxyPT)* operation eliminates the “proxy c” element that might have been generated by the *Extend(proxyPT)* operator. The same result can be achieved by the *Reduce(“real subject”, proxyPT)* operator. Recalling the example for

the *Extend(P)* operator, such situation may be useful to represent the returning of a mobile agent/object (i.e. the “real subject”) back in the chain to the last visited place.

The second version of the *Reduce* operator is also used in Figure 4.10 to eliminate one of the inner members that is annotated both as a “proxy” to the next element in the chain and as the “real subject” to the previous element in the chain. Concretely, the *Reduce(“proxy c”, proxyPT)* operator eliminates the previous proxy with a dual role in the chain, i.e. the “proxy b/real subject a” is deleted and the element “proxy c/real subject a” element is now the surrogate of the “real subject” for the element “proxy a”. An example of such situation, may be the moving of “proxy c” to its previous location, and the consequent elimination of “proxy b” (which becomes unnecessary).

Finally, and recalling the extension of an ordinary proxy into a surrogate for a newly created proxy, i.e. the (third) example on bottom of Figure 4.9, such transformation can be undone by applying the *Reduce* operator to the first member in the proxy chain that is annotated with a dual definition. Specifically, the *Reduce(“proxy b”, proxyPT)* operator eliminates the ordinary proxies connected to “proxy b” (in this case, only “proxy a”), and transforms the element “proxy b/real subject a” into an ordinary proxy (“proxy b”).

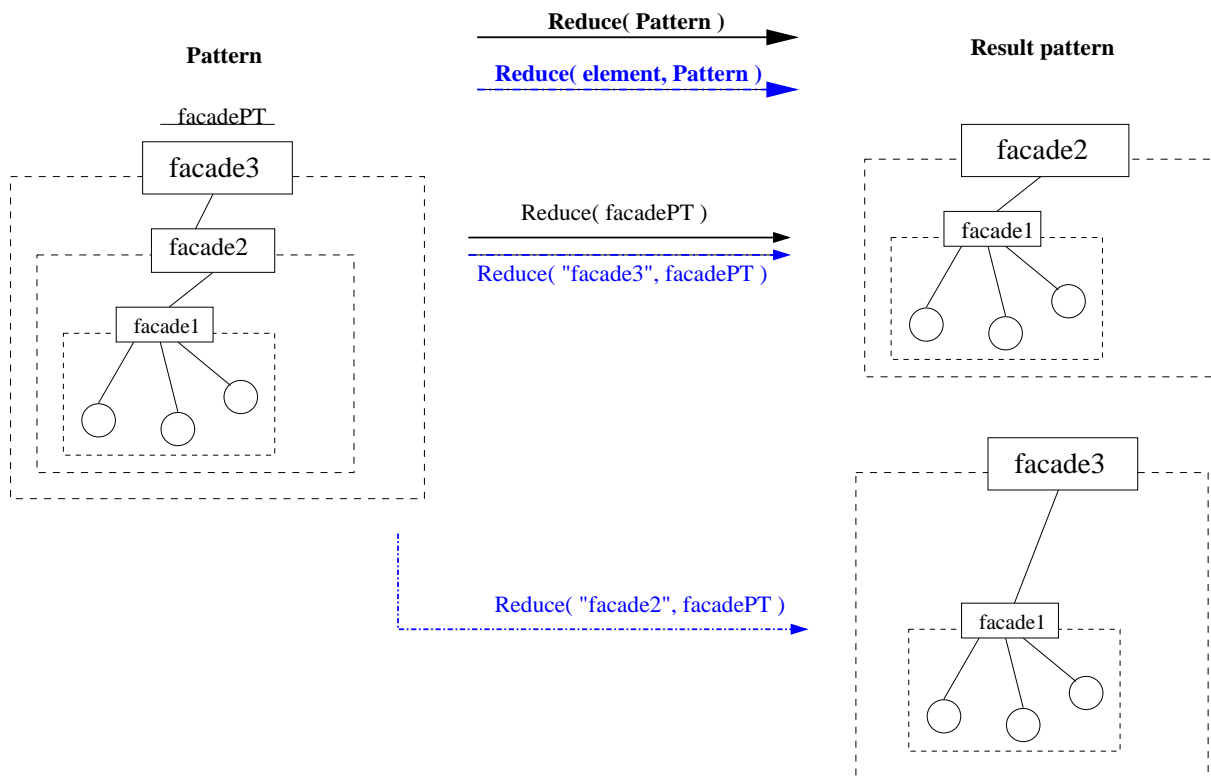


Figure 4.11: Examples of both versions of the *Reduce* operator over one Facade Template.

Figure 4.11, in turn, presents the application of both versions of the *Reduce* to a *Facade S-PT*. The manipulations *Reduce(facadePT)* and *Reduce(“facade3”, facadePT)* produce an equal result, namely the elimination of the outmost Facade represented by the “facade3” element. On the other hand, the *Reduce(“facade2”, facadePT)* eliminates that inner Facade (i.e. “facade2”). Considering the example of the portal for a Grid domain, such deflations of the structure of the pattern “facadePT” would eliminate the created extensions, in case the access to new or other

types of services was to be provided only temporarily, i.e. the original Facade interface should be restored after that time.

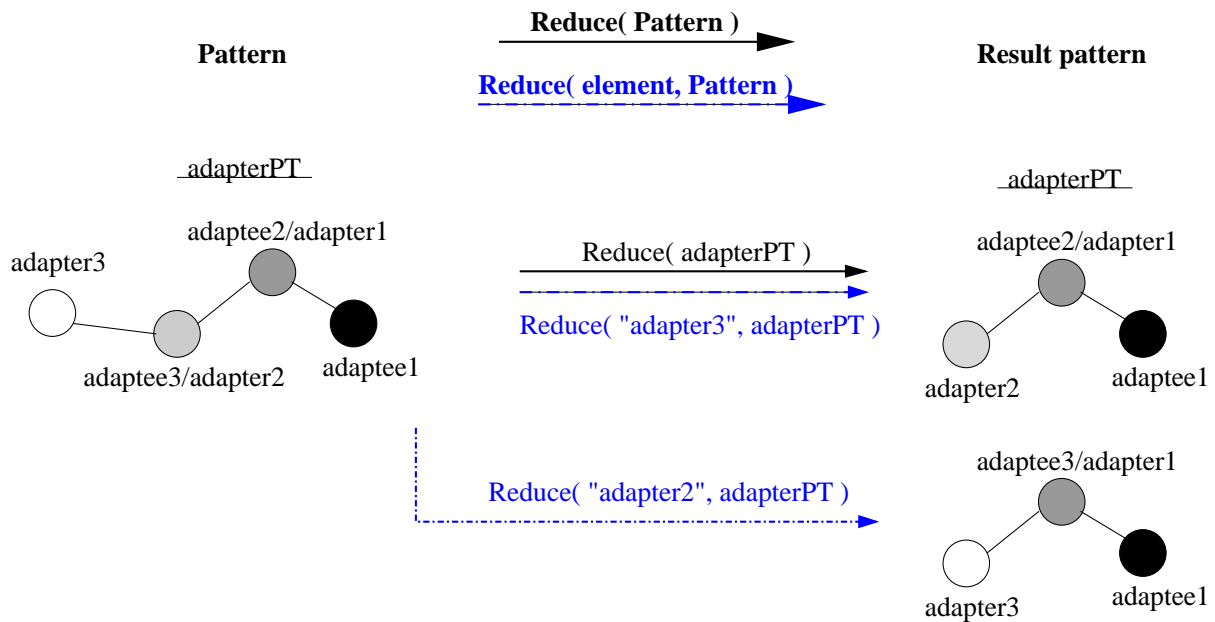


Figure 4.12: Examples of both versions of the *Reduce* operator applied to an *Adapter Template*.

Finally, examples of the application of both versions of the *Reduce* operator to an *Adapter* pattern are presented in Figure 4.12. The manipulations *Reduce(adapterPT)* and *Reduce("adapter3", adapterPT)* generate the elimination of the last adapter, i.e. "adapter3". However, the *Reduce("adapter2", adapterPT)* eliminates the intermediate adapter "adaptee3/adapter2". Considering the example of the usage of the *Adapter* pattern to support the interface to a legacy code, a deflation of the structure through several calls to *Reduce* may, for example, eliminate the outmost and the inner adapter elements (e.g. "adapter3" and "adapter2" in the Figure), allowing the reuse of the pattern to yet another environment that would simply require the original adaptation (i.e. "adapter1").

4.2.2 Grouping Operators

This section describes the semantics of the *Group* and *Embed* operators which support the formation of *Hierarchical Patterns*, and their associated operators, namely *Ungroup* and *Extract*, respectively.

Group and Ungroup

The semantics of the *Group(P1, ..., Pn, ResultP)* and *Ungroup(P)* operators is quite simple. All types of S-PTs can be aggregated into a group template that will, thereafter, represent all its members as a whole. This resulting group template is one example of a *Hierarchical Pattern* (i.e. it contains other pattern templates). The single structural relationship among the S-PTs belonging to the newly formed group template is that they are represented and accessed as a

single entity. The identifier of this new pattern is defined in the call of the *Group* operator by the parameter “ResultP”.

Such grouping operator may be useful, for example, to aggregate a set of patterns and, subsequently, apply one of the previously described *Ownership* operators. This permits defining access restrictions to the group, and consequently, to its members. Group templates may also ease the process of mapping an application configuration onto the lower level resource management layers. Moreover, a group may be replicated through the *Replicate* operator, thus avoiding the need to duplicate its members individually. At any time, a group may be undone through the *Ungroup* operator.

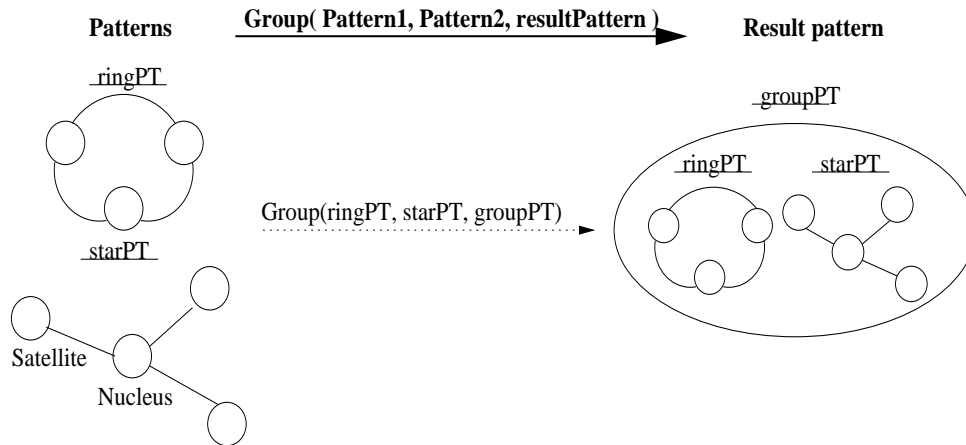


Figure 4.13: The pattern templates “ringPT” and “startPT” are grouped through the *Group* operator, and the resulting aggregate is named “groupPT”.

Figure 4.13 depicts the grouping of a *Ring* S-PT and a *Star* S-PT into an aggregation named as “groupPT”. As a result, both S-PTs are said to be at the same “level” within the group, specifically the first level of the group, and this one is said to have a *cardinality* equal to two (i.e. “groupPT” has two members). Furthermore, an aggregation does not have the concept of “position” since there is no structural associations among the group’s members. However, it is possible to access the members of the group by concatenating the group’s name and a member’s name. For example, the “ringPT” in Figure 4.13 is accessible through the identifier “groupPT.ringPT”, and the access to the other member is done through the identifier “groupPT.starPT”. Appropriately, the elements within the structure of these enclosed patterns can also be accessed by concatenating the identifier of the pattern with the identifier of its element. For example, the identifier “groupPT.starPT.nucleus” would allow the access to the “nucleus” element within the “starPT”.

Figure 4.14, in turn, shows the disaggregation of the aggregate “groupPT” through the *Ungroup* operator and, as presented in the Figure, the ex-members are not deleted. Even if the members of the group passed as argument to the *Ungroup* operator are groups themselves, they remain intact after the manipulation, i.e. the *Ungroup* operator is not recursive. The possibility of enclosing one group into another will be described in the discussion of the *Embed* operator in a section ahead.

To further clarify the *Group* operation, Figure 4.15 shows the result of adding a new S-PT to an existing group PT. Namely, the pattern “facadePT” is added to the “groupPT”, and the

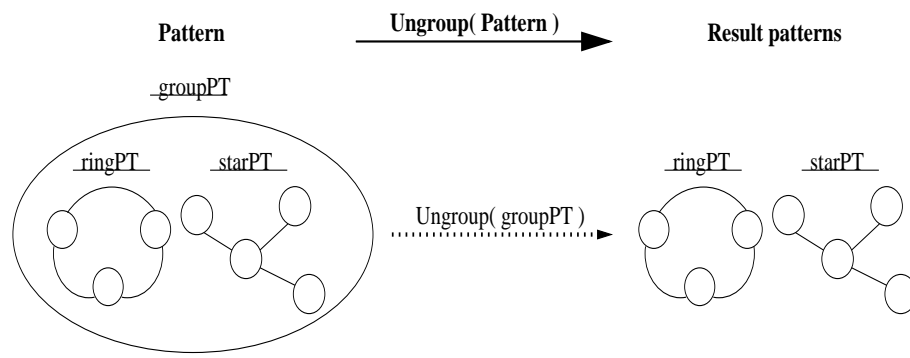


Figure 4.14: The group "groupPT" is dissolved through the Ungroup operator.

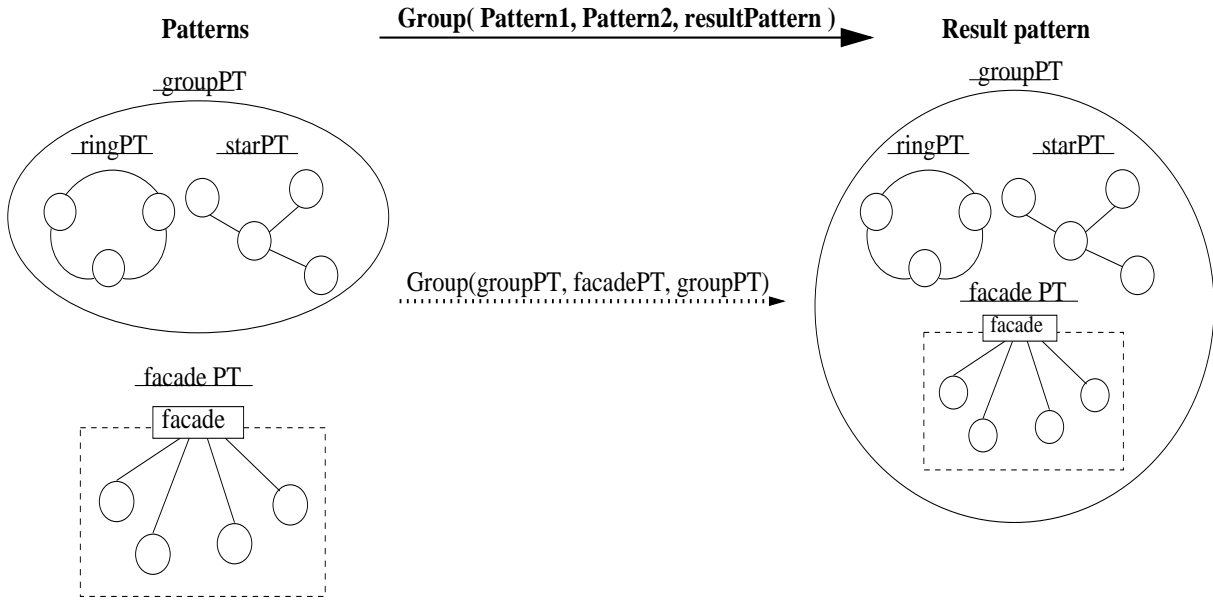


Figure 4.15: Adding a extra pattern template to the aggregate "groupPT".

cardinality of the group increases to three. Please not that in case of adding a pattern to a group, if that pattern's identifier already exists within the group, its name will be changed somehow to guarantee that all group member's at the same level have a unique identifier within the group.

Additionally, Figure 4.16 shows a case of aggregating two groups, namely "group1PT" and "group2PT". In consequence of this operation, the two groups are merged into one named "group1PT", and the identifier "group2PT" disappears since the label "group1PT" was indicated as the name for the result group.

Embed Operator

According to the semantics of the *Embed(P1, P2, position)* operator, the pattern identified by parameter "P1" is to be embedded into the destination pattern identified by the parameter "P2" by instantiating a particular component place-holder within "P2". That specific component place-holder is fully determined within the structure of pattern "P2" through the parameter "position", which typically refers to the identifier of the component place-holder.

As a result of the *Embed* operation, the pattern "P1" becomes one of the elements within the

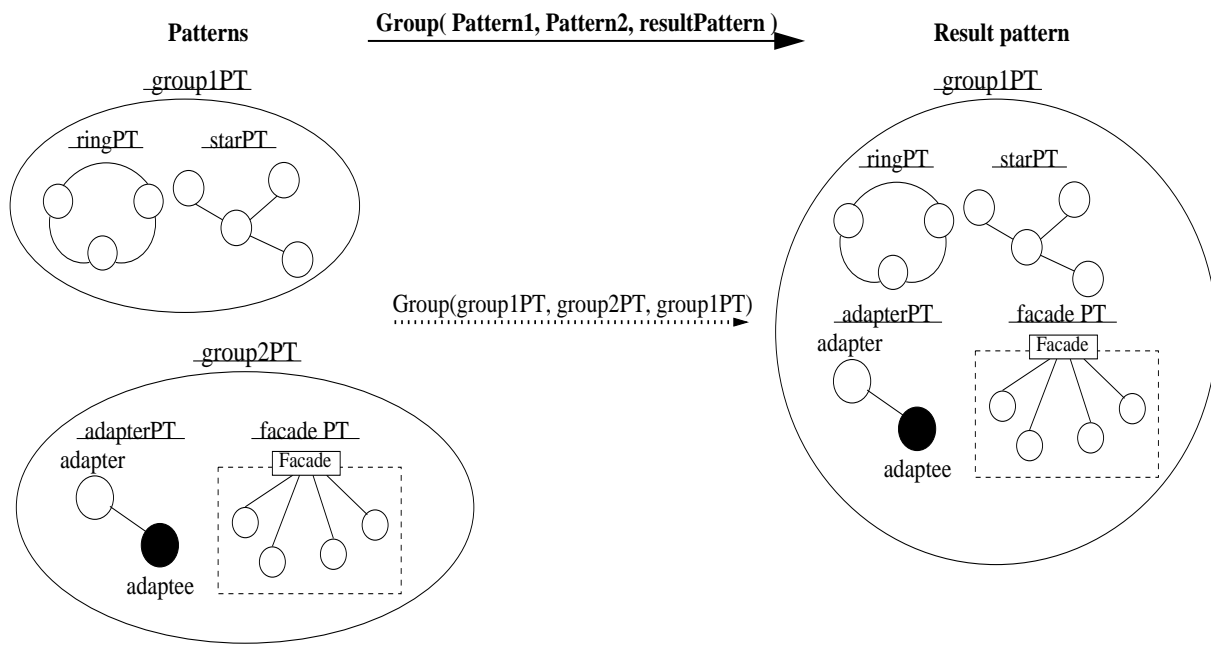


Figure 4.16: Merging of groups "group1PT" and "group2PT" through the Group operator, producing the aggregate "group1PT".

structure of pattern "P2". Consequently, pattern "P2" is now classified as a *hierarchic pattern*, i.e. a pattern with one or more patterns as its structural elements. Clearly, the *Embed* operator only succeeds if the parameter "position" identifies a free component place-holder.

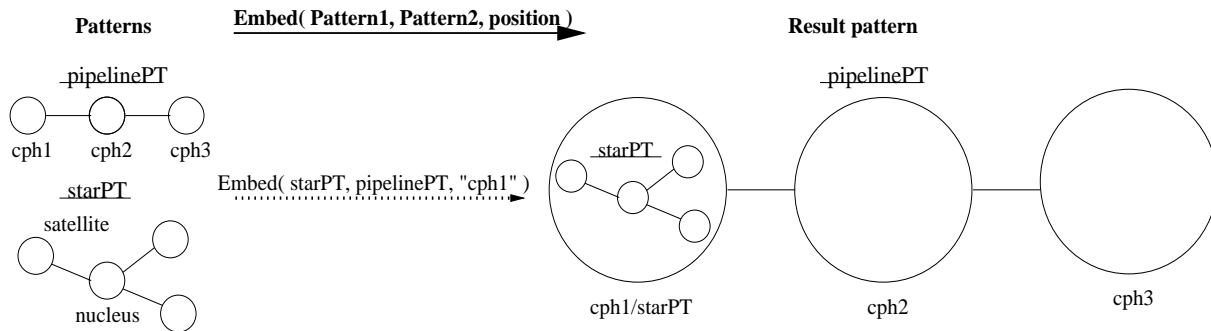


Figure 4.17: An example of a pipeline template with an embedded pattern (a star) in the left-most stage ("cph1"). This hierarchic pattern template is built through the Embed operator.

For example, Figure 4.17 represents the embedding of a *Star* S-PT, i.e. "starPT", into the first stage ("cph1" element) of a *Pipeline* S-PT, i.e. "pipelinePT". The first stage of "pipelinePT" keeps its original identifier, i.e. "cph1" but it is also identified by the name of the embedded pattern, i.e. "starPT". The embedded pattern can thereafter be accessed through the identifier "pipelinePT.starPT", but its position is also defined as "pipelinePT.cph1". The cardinality of this resulting *Hierarchical pattern* remains equal to three, since the number of stages of the "pipelinePT" did not change. Therefore, the cardinality of a *Hierarchical pattern* refers to the number of elements (with other embedded patterns or not) within the first level of that pattern.

The embedding operation is useful, for example, when combining different subsystems in a Grid environment. The user may start by first defining a pipeline S-PT to represent a sequence

of Grid services and tools. This may be the case of a scientific application (head of the pipeline) that generates results for a data analysis tool, which in turn produces data to a visualisation tool (corresponding to the last stage of the pipeline). A user familiar with the structure of the problem to be solved may then define the scientific application's configuration with another pattern. For instance, considering that the scientific application is computationally intensive, the user may model its configuration through a star topology supporting a central manager – perhaps running on a parallel machine or high-end server, and a number of sub-servers that interact with it. Assuming, for instance, that the behaviour of this sub-system follows the Master/Slave pattern, this behaviour can then be developed over the star topology. Hence, the user creates a new star S-PT (with an adequate number of satellites for supporting the slaves), and embeds this S-PT in the first position of the pipeline S-PT, thus producing a hierarchical structure.

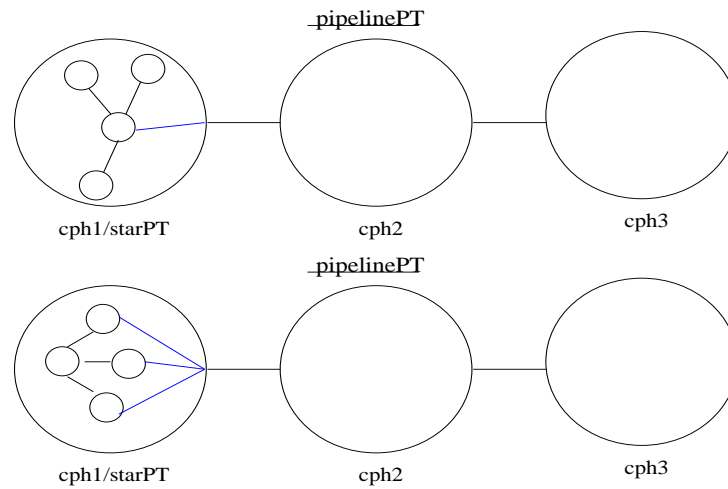


Figure 4.18: Examples of possible connections between the embedded pattern and the enclosing pattern.

To conclude, please note that the *Embed* operation does not define which elements of the “StarPT” in the example in Figure 4.17 are to be structurally connected to the enclosing CPH (i.e. “cph1”) which, in turn, is structurally connected to “cph2” according to the pipeline definition. Specifically, the *Embed* operator does not define if, as exemplified in Figure 4.18, a structural connection (that will represent a data/control flow within a Behavioural Pattern) is to be made only between the “nucleus” of “StarPT” and the enclosing “cph1”; or, if all satellites should be connected to “cph1”; etc.

We consider that the way such definition is made is implementation dependent. For example, in our implementation onto the Triana PSE, the components/tools/services that instantiate the component place-holders have input/output ports. Moreover, a component place-holder which encloses a pattern is supported by a *group* in Triana, and this one is defined as a component as well, i.e. a group may also have input/output ports. The Triana GUI allows making direct connections between those ports, but our implementation also defines by default, and for all implemented patterns, how one embedded pattern is connected to the other elements in the enclosing pattern. However, and to avoid such restrictions, we recognise the need to include in the model the possibility of defining direct connections between elements, so that a

complete configuration can be built from a script.

Extract Operator

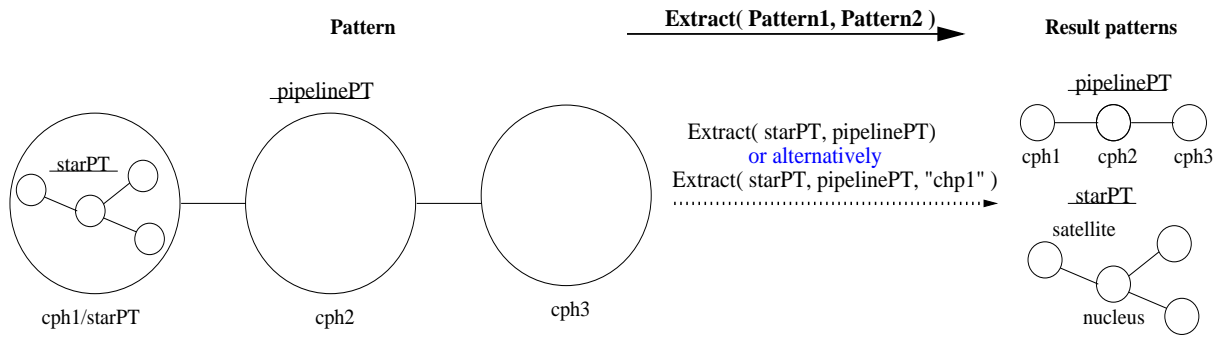


Figure 4.19: Extracting pattern “starPT” from within the first stage of the pattern “pipelinePT”.

The complementary operator to the *Embed*, namely *Extract*(*P1*, *P2*, *position*), removes pattern “P1” from a specific location within pattern “P2” which is identified by the parameter “position”. Figure 4.19 presents the result of the particular manipulation *Extract*(*starPT*, *pipelinePT*, “cph1”). Considering that each pattern template has its own unique identifier within the enclosing pattern, the “position” argument may be omitted from an *Extract* call. The second version of this operator, which does not require the parameter “position”, can therefore be used to produce the same result, i.e. *Extract*(*starPT*, *pipelinePT*), as presented in the same Figure.

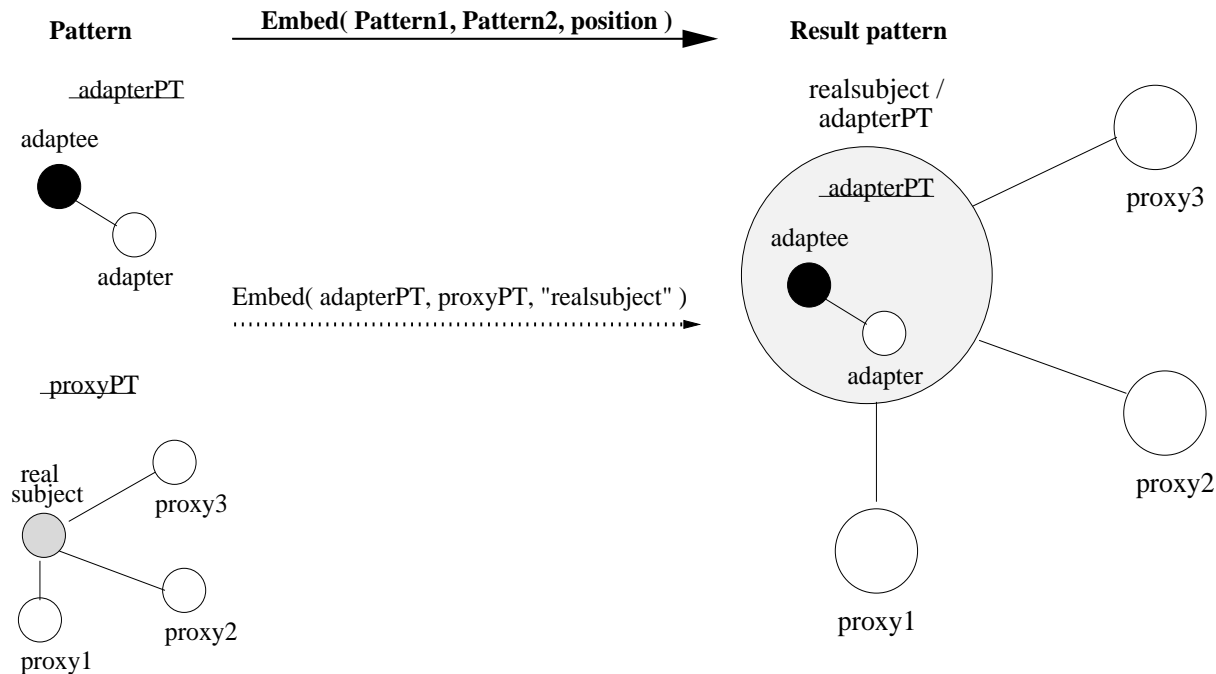


Figure 4.20: Embedding an adapter template into a proxy template in the position of the “real subject”.

Another example of applying the *Embed* operator is shown in Figure 4.20, where an *Adapter*

S-PT ("adapterPT") becomes the "real subject" of a *Proxy* S-PT ("proxyPT"). Such configuration may be useful, for instance, to provide access to a Grid Service for different types of users, each one with distinct access policies and request types. Dissimilar "protection proxies" interface the users to the service, with this one requiring adaptation to attend the different types of requests.

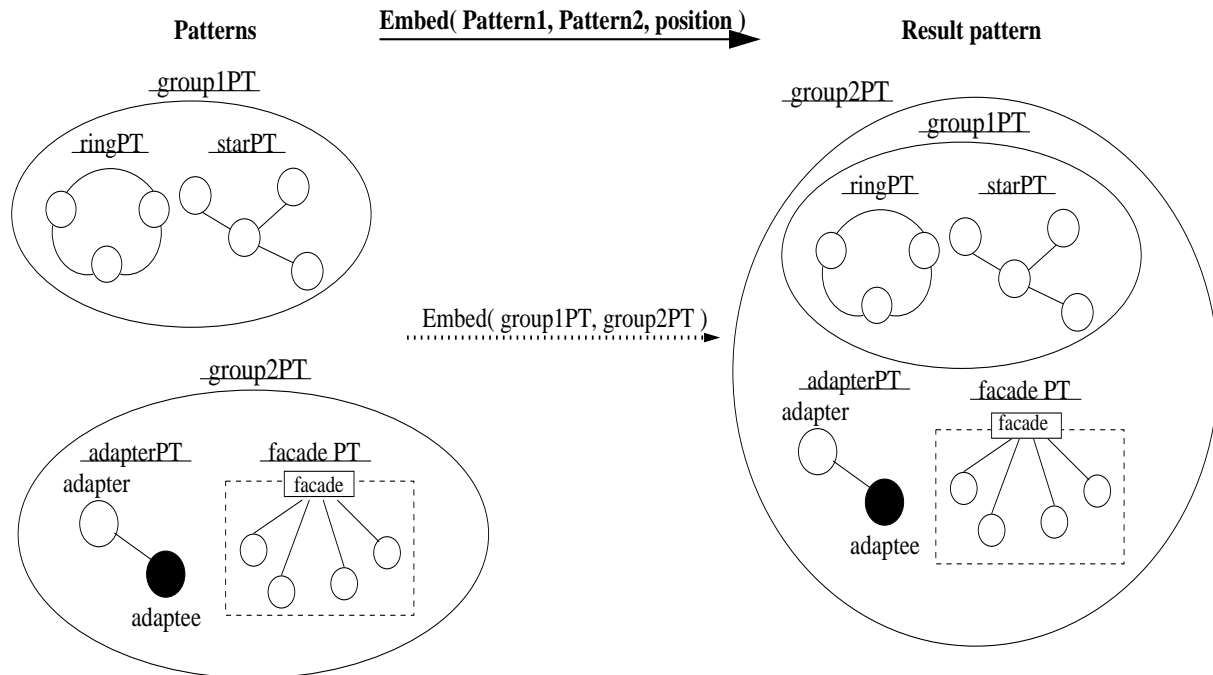


Figure 4.21: Embedding a group ("group1PT") into another ("group2PT").

To clarify the *Embed*'s semantics when applied to group templates, Figure 4.21 illustrates the embedding of the aggregate "group1PT" into another group, namely "group2PT". Such operation can be accomplished by the version of the *Embed* operator that does not require a parameter for the embedding position, i.e. *Embed(P1, P2)*. Comparing with the aggregation of two groups, which results in a single group containing all the elements (as was depicted in Figure 4.16), the *Embed(group1PT, group2PT)* operator implies that the embedded group, i.e. "group1PT", becomes a sub-group of the encloser group, i.e. "group2PT". As such, the result of the *Embed* is also a hierarchy.

Finally, the embedding of a PT that is not a group into a group template has an equivalent semantics to the aggregation of an extra PT to the original group (this example was shown in Figure 4.15).

The following discussion of embedding a pattern template into a *Hierarchical Pattern Template* completes the description of the *Embed* operator's semantics.

Hierarchical Pattern Templates

A *Hierarchical Pattern Template* is either: 1) a pattern template with one or more embedded pattern templates (the embedded patterns may themselves have other embedded pattern templates); or 2) a group template. In the latter case, a group may also contain other sub-groups, and both the outmost group and the inner groups may also contain higher-level templates as

defined in point 1). The semantics of a *Hierarchical PT* is therefore recursive resulting on a hierarchy of enclosed pattern templates which is built through the *Embed* operator (or also through the *Group* operator in the case of a group template).

The enclosed patterns in the *Hierarchical Pattern Template* are directly accessible, meaning they can be manipulated individually by the operators. Depending on the operator, the access to those inner patterns can either be based on their unique identifiers within the pattern of whom they are members, or alternatively, on the ordered concatenation of the names of all the pattern templates that enclose them. In this case, the name of the operated (inner) pattern comes last in that sequence, and the name of the outmost pattern comes first.

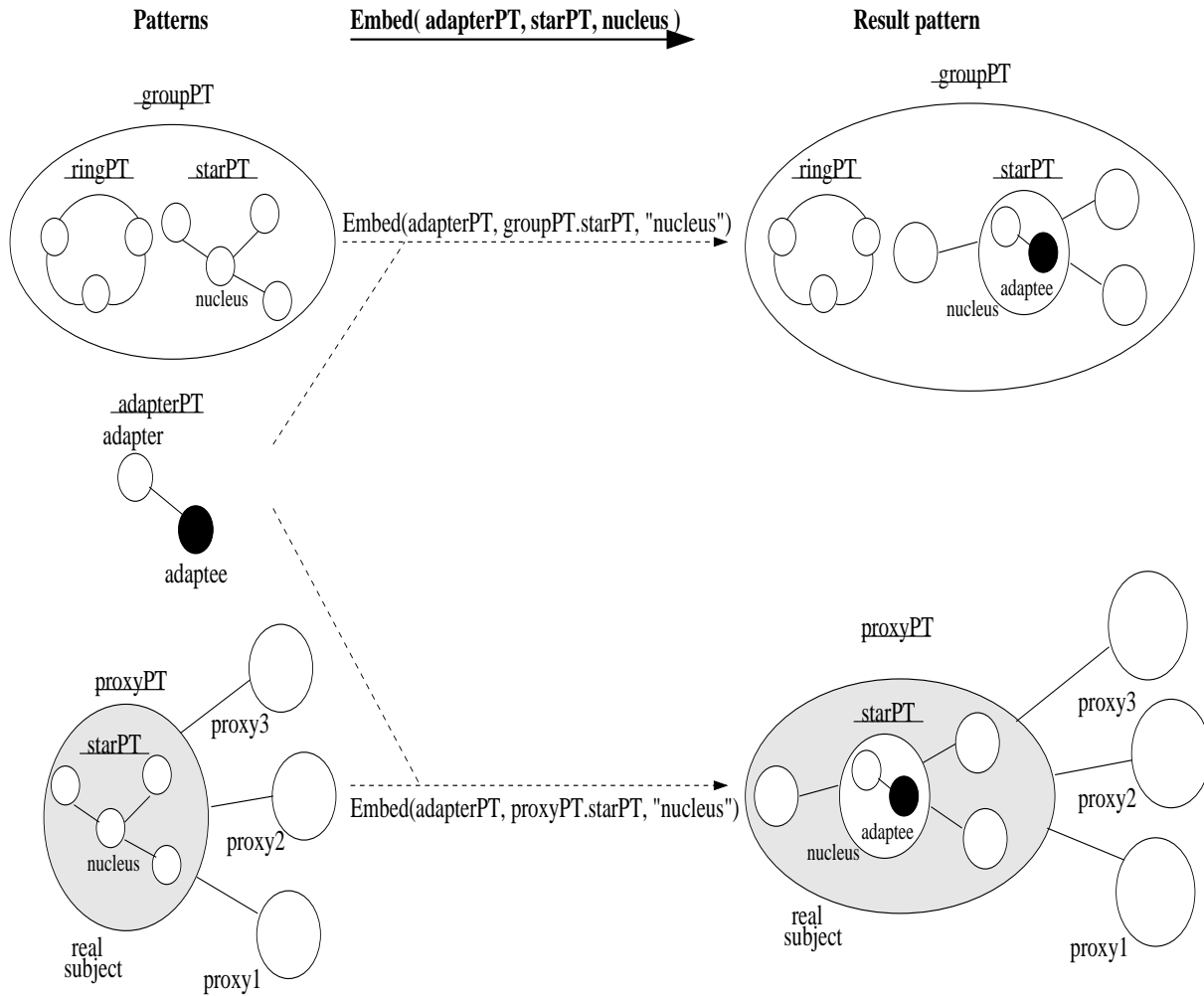


Figure 4.22: Embedding the same pattern template into two Hierarchical Pattern Templates. In both examples, the "adapterPT" is embedded in the "nucleus" of a "starPT", but in one case (upper part of the Figure) this latter pattern is included in a group, whereas in the other, the "starPT" is embedded in the "real subject" position of the "proxyPT".

The left side of Figure 4.22 shows two Hierarchical PTs, namely, a group pattern template ("groupPT") and a Proxy template ("proxyPT") with an embedded Star template in the "real subject" position ("starPT"). The Figure exemplifies how to embed a pattern into those two kinds of Hierarchical PTs in well specified positions. In the example, the same pattern template, namely "adapterPT", is embedded into the "nucleus" of the "startPT" which is present in both Hierarchical PTs. In both *Embed* examples, the latter template is referenced by name

concatenation, concretely, "groupPT.starPT" in the case of the group PT, and "proxyPT.starPT" in the case of the Proxy PT.

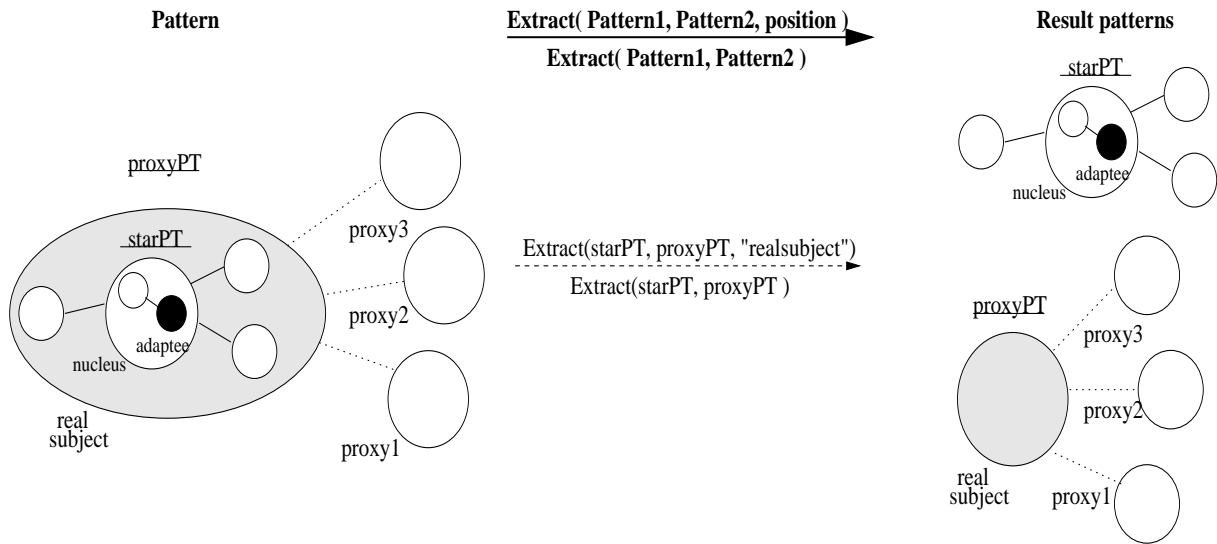


Figure 4.23: Extracting a pattern template from a Hierarchical Pattern Template, namely a *starPT* is removed from within the "real subject" of a *proxyPT*, and the "real subject" gets uninstantiated.

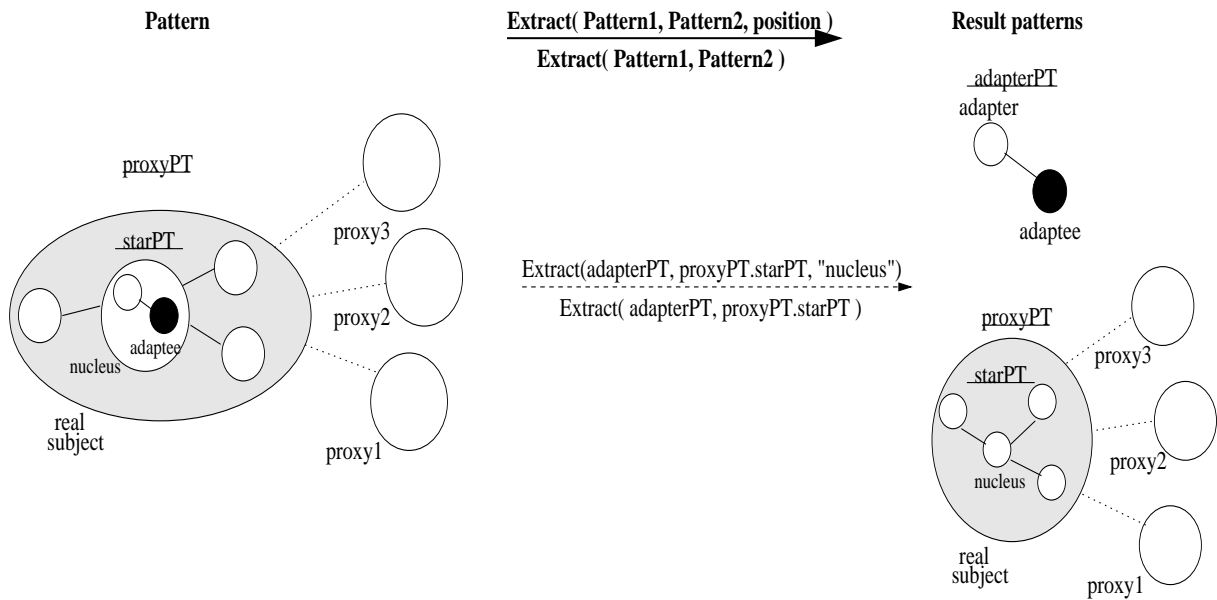


Figure 4.24: Extracting a pattern template from a Hierarchical Pattern Template, specifically, the *adapterPT* is extracted from the "nucleus" of the *starPT* which is located in the position of the "real subject" in the *proxyPT*. Consequently, the "nucleus" of the *starPT* gets uninstantiated.

The extraction of patterns from a Hierarchical PT is illustrated in both Figures 4.23 and 4.24. In the first case (Figure 4.23), a pattern template is removed from the first level of the hierarchy, namely, the "starPT" template is extracted from the "real subject" of the "proxyPT" by the *Extract(starPT, proxyPT, "real subject")*. Consequently, the "real subject" becomes uninstantiated

and, later on, another pattern can be embedded in that component place-holder. Considering that the “starPT” identifier is unique within the “proxyPT” pattern, the *Extract(starPT, proxyPT)* produces the same result.

In the second case (Figure 4.24), the pattern to be removed, namely “adapterPT”, is located in the second level of the hierarchy. Specifically, “adapterPT” is located in the “nucleus” of the “starPT” which, in turn, is enclosed by the “proxyPT” in the “realsubject” position. After the removal of the “adapterPT” by the *Extract(adapterPT, proxyPT.starPT, “nucleus”)* operator, the “nucleus” of the “starPT” stays uninstantiated. As can be seen by this second example, after the extraction, the “adapterPT” moves to the same level as the outmost pattern, i.e. “proxyPT” preserving, in this way, the structural consistency of this latter pattern. Considering also here that the “adapterPT” identifier is unique within the “proxyPT.starPT” pattern, the second version of the *Extract* operator, i.e. *Extract(adapterPT, proxyPT.starPT)* generates the same result.

Please note that in the case of group-based Hierarchical PTs, the *Extract* and *Embed* operators can be used to move PTs between those inner sub-groups without loss of structural consistency, according to the semantics of group-based pattern templates.

4.3 Sequences of Structural Operators

Structural Operators can be applied in sequence and some can be composed forming *Compound Structural Operators*. This sub-section aims to illustrate some relevant examples of those situations.

4.3.1 Sequences Including the Replicate, Replace, or Reshape Operators

The present examples aim to further illustrate the utility of the *Replicate*, *Replace*, and *Reshape* operators.

First, if applied to a *Hierarchical Pattern Template* produced by the *Group* operator, the *Replicate* operator allows a faster way to duplicate a set of PTs. For example, the next sequence defines a possible way of duplicating a set of patterns:

```
Group( P1, P2, P3, P4, P5, groupPT )
Replicate( 1, groupPT, ``group2PT`` )
Ungroup( groupPT )
Ungroup( group2PT)
```

According to the semantics of the *Replicate* operator previously discussed, the application of this operator to the “groupPT” pattern in the previous sequence, results in the creation of a clone with a distinct identifier, namely, “group2PT” in the example. The subsequent disaggregation of the two groups through the *Ungroup* operator results in the original patterns plus their clones (each one having its own unique identifier as well).

Second, the *Replicate* operator can be used to build a clone of a more complex pattern template like “proxyPT”, as shown in Figure 4.25. The “proxyPT” was previously presented in Figure 4.22. The slightly different pattern to be built (“proxy2PT”) is dissimilar to “proxyPT”

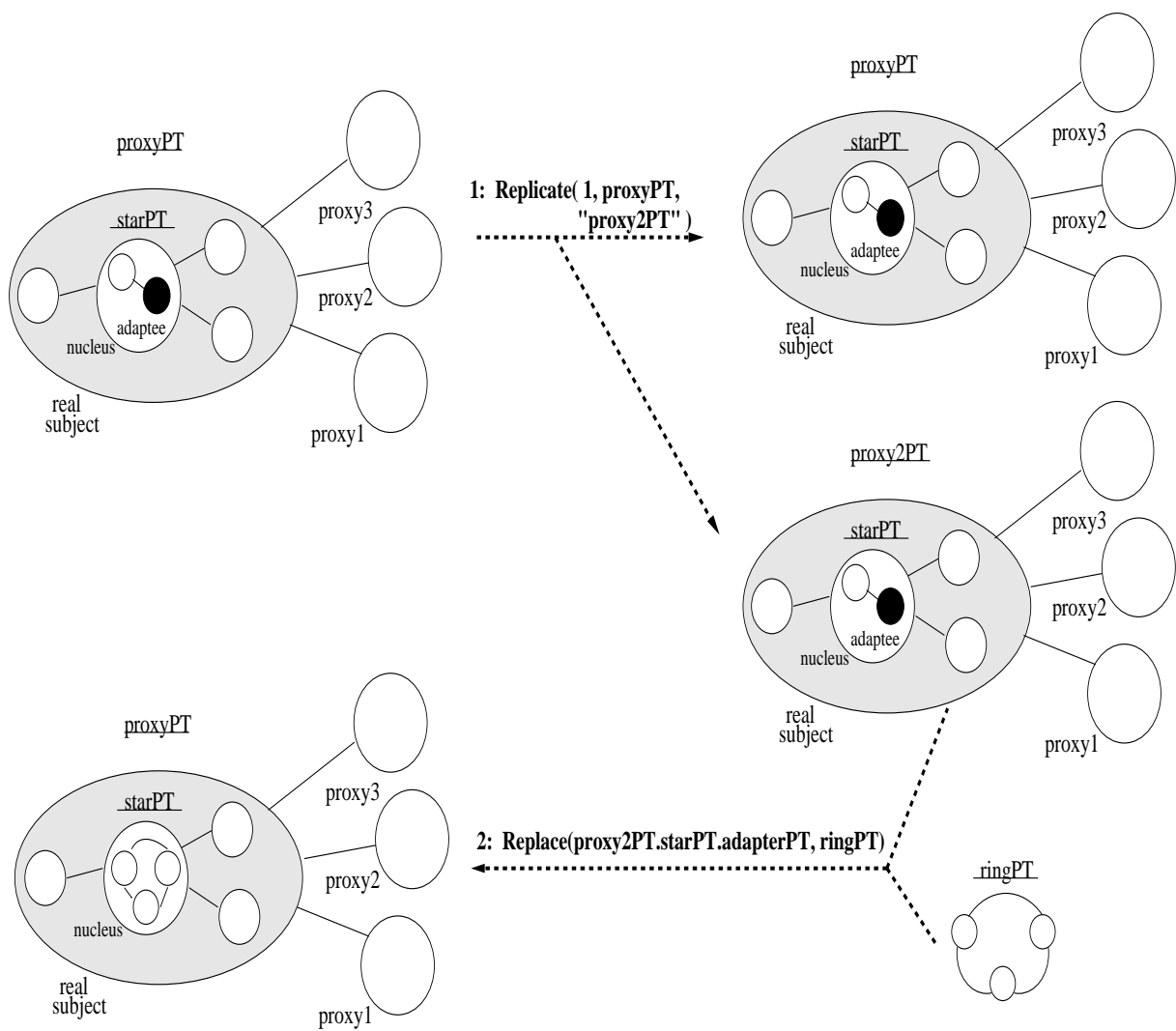


Figure 4.25: Building a new similar PT to the ProxyPT but where a ringPT is embedded in the "nucleus" of the starPT (instead of the adapterPT).

in the sense that a diverse pattern should be embedded in the "nucleus" of the "proxyPT.starPT" instead. As shown in Figure 4.25, such result is obtained by applying the *Replace* operator to the "adapterPT" in "proxy2PT", substituting it by the "ringPT". An example of an operator sequence to build such result might be:

```
Embed( Embed( adapterPT, starPT, nucleus ), proxyPT, realsubject ) )
Replicate( 1, proxyPT, 'proxy2PT' )
Replace( proxy2PT.starPT.adapterPT, ringPT )
```

The previous sequence also gives an example of the *Embed* operator composed with itself, forming one case of a *Compound Structural Operator*.

Finally, the *Reshape* operator can also be directly applied to a pattern embedded into a *Hierarchical Pattern Template*, as long as the restrictions defined in sub-section 4.2.1 are obeyed. For example, Figure 4.26 shows the transformation of the "starPT" (embedded in the "real subject" of the "proxyPT") into a ring pattern template ("ringPT").

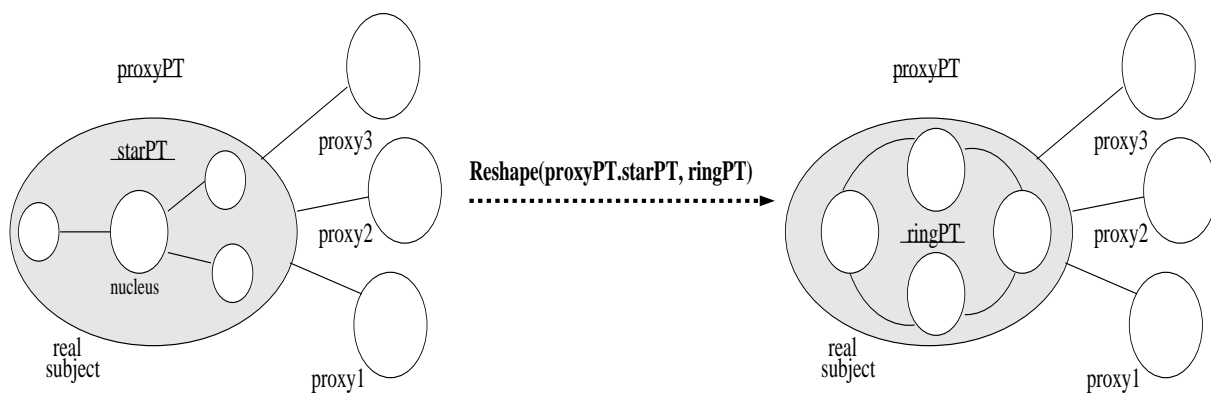


Figure 4.26: Transforming an embedded pattern into another through the Reshape operator.

4.3.2 Sequential Application of Extend, Increase/Decrease, and Reduce

The next examples aim to provide a clarification of structural operator sequences including the *Increase*, *Decrease*, *Extend*, and *Reduce* operators. First of all, a sequence of those operators is only applicable to Proxy and Facade S-PTs since, on one hand, the *Extend/Reduce* operators cannot be applied to the *Topological Structural Patterns* (i.e. Ring, Star, and Pipeline) and, on the other hand, the *Increase* operator cannot be applied to the Adapter pattern (this was previously explained in sub-section 4.2.1).

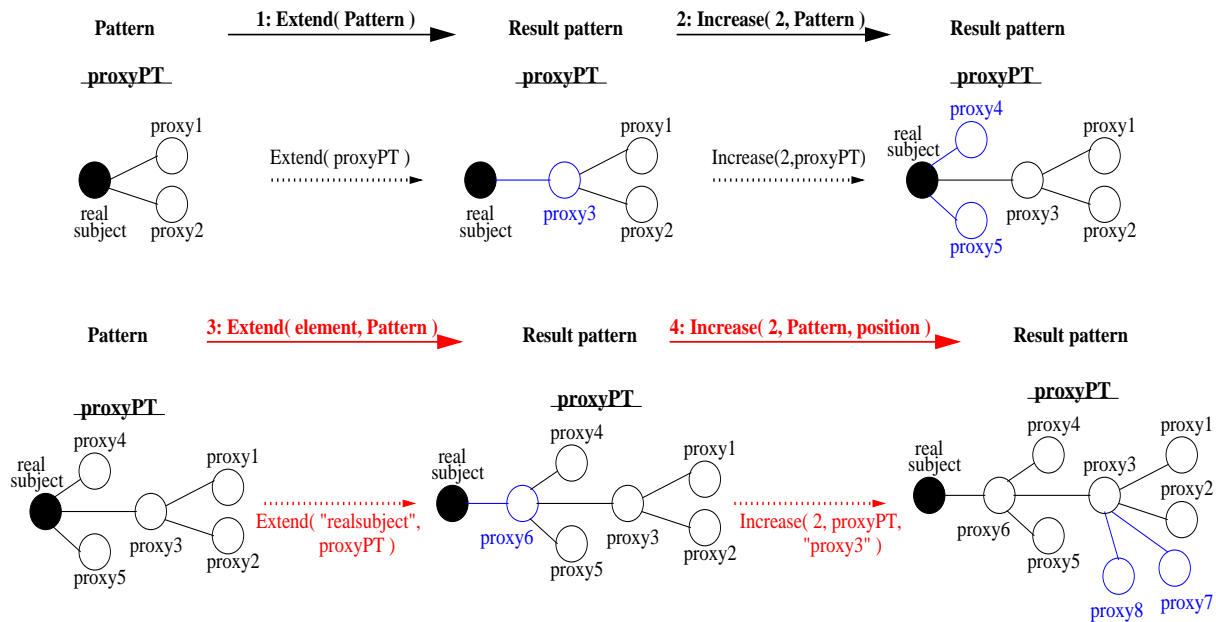


Figure 4.27: Applying the Extend and Increase operators in sequence to a Proxy PT.

As a first example, Figure 4.27 represents the sequential manipulation of a *Proxy* S-PT, i.e. “proxyPT”, by the *Extend* and *Increase* operators. On the top of the Figure, the “proxyPT” is manipulated by the first versions of those operators, namely *Extend(proxyPT)* and *Increase(2, proxyPT)* (steps 1 and 2). The result of these manipulations is, respectively, the creation of “proxy3” that is also annotated as a *Real Subject* for the “proxy1” and “proxy2” elements, and

the creation of two new ordinary proxies associated to the “realsubject” element, i.e. “proxy4” and “proxy5”. These actions can be represented by the following sequence:

```
Extend(
proxyPT ) Increase( 2, proxyPT )
```

which could also be defined through a *Compound Structural Operator* like

```
Increase( 2, Extend( proxyPT ) )
```

On the bottom of Figure 4.27, the pattern “proxyPT” is subsequently manipulated by the second versions of the *Extend* and *Increase* operators (steps 3 and 4). First, the *Extend*(“realsubject”, proxyPT) operator generates the new “proxy6” element that becomes its local surrogate for the pre-existent proxies, i.e. “proxy4” and “proxy5”. Second, the *Increase*(2, proxyPT, “proxy3”) operator creates two ordinary proxies associated to the “proxy3” element. These two actions can also be represented by a sequence:

```
Extend(
``realsubject``, proxyPT ) Increase( 2, proxyPT, ``proxy3`` )
```

We recall that the selected members to instantiate the “position” parameter in the *Increase* (*n*, *P*, *position*) operator have to be consistent with the *Proxy* pattern’s definition. Specifically, the eligible members are the ones which are (also) defined as a *Real Subject* to the ordinary proxies to which it is directly associated. Therefore, it was possible to increment the number of proxies of the member “proxy3”, as it would be to instantiate the “position” parameter with the member “proxy6” and, of course, “realsubject”.

As a second example, a corresponding manipulation of a *Facade* S-PT by an interleaved application of both versions of the *Extend* and *Increase* operators is presented in Figure 4.28. The four operations in the Figure can be represented by the following operator sequence:

```
Extend( facadePT )
Increase( 2, facadePT )
Extend( ``facade1``, facadePT )
Increase( 2, facadePT, ``facade3`` )
```

With the first operator, the structure of the “facadePT” is extended resulting on the creation of a new “facade” element within the structure, i.e. “facade2” in the Figure. The second operation increases the number of elements of the extended “facadePT”. Since the position of where to create the new sub-systems is not explicitly defined, the *Increase*(2, facadePT) creates the two new CPHs on the outmost facade element by default, namely within the structural context of “facade2” element. Subsequently, the “facadePT” is again extended through *Extend* but with the explicit description where to augment recursively the structure. Concretely, the *Extend*(“facade1”, facadePT) operator (step 3) generates a new Facade structure between the pre-existent Facade structures, i.e. the “facade3” element provides an interface for the “facade1” element and other sub-systems that may be created within that new Facade structure. Such may be represented by the *Increase*(2, facadePT, “facade3”) operator (step 4) that creates two new CPHs for two sub-systems associated to the “facade3” element. We again recall that the *Increase*(*n*,

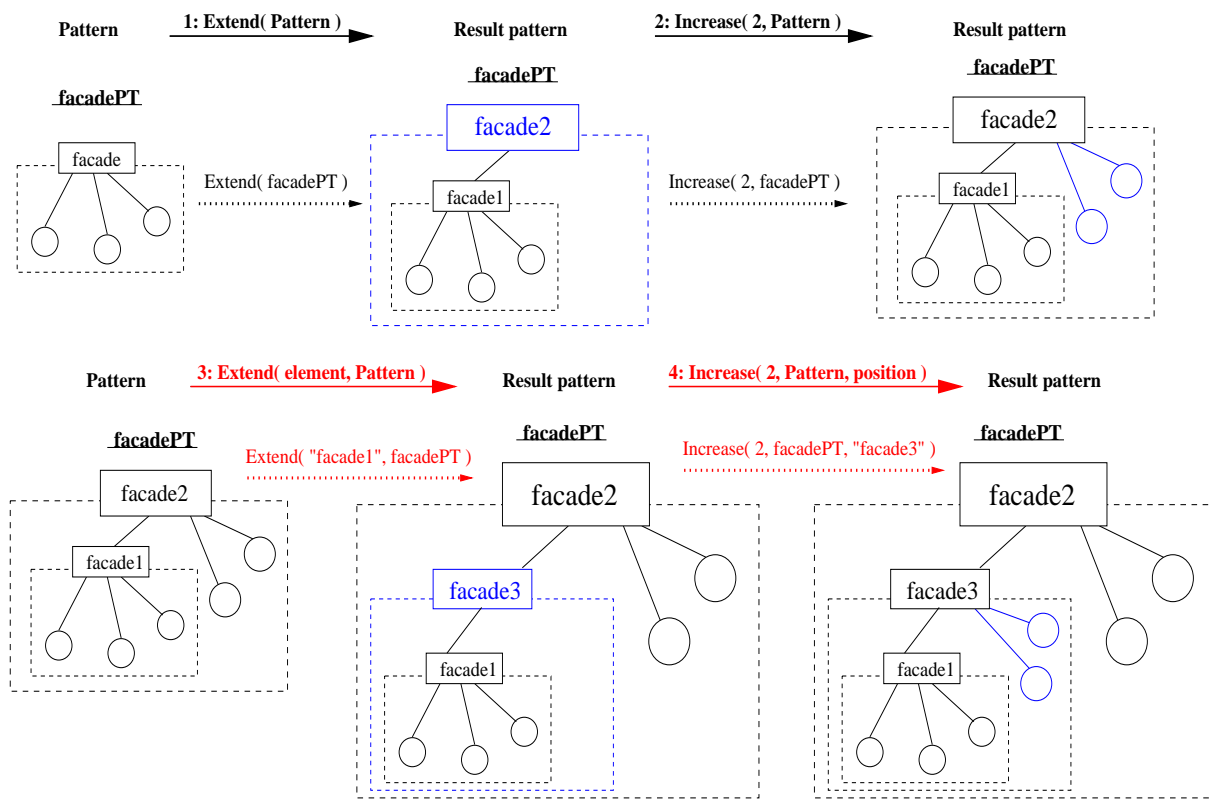


Figure 4.28: Applying the *Extend* and *Increase* operators in sequence to a Facade PT.

facadePT, position) can only be applied to the “facade1”, “facade3”, and “facade2” elements within the “facadePT” pattern.

As for similar operator sequences like the ones above which include the *Increase/Decrease* operators, but where the *Reduce* operator is used instead of *Extend*, their results are somehow symmetric to what was described for the latter operator. Specifically, since the *Reduce* operator undoes a corresponding *Extend* operation, the result of applying the *Reduce* is as if the *Extend* operator had not be applied at all.

Figures 4.29 and 4.30 present examples of the *Reduce* operator to cases of *Proxy* and *Facade* S-PTs. In the first Figure, both versions of the *Reduce* are applied to the “proxyPT”. First, the *Reduce(proxyPT)* operator, which is similar to the *Reduce(“realsubject”, ProxyPT)*, deflates the previously extended pattern in a pre-defined way. Concretely, the reduce operation is applied to the element annotated as the *Real Subject* within the pattern and, as a result, the “realsubject” element takes the place of the “proxy6” element within the structure. Therefore, the “proxy7” element becomes incoherent and is deleted, and the “proxy4” and “proxy5” elements are now associated to the “realsubject” as a result of the deletion of “proxy6”.

Second, the *Reduce(“proxy6”, proxyPT)* operator defines that the reduction of the structure of “proxyPT” is to be performed at the “proxy6” element, meaning that this element will take the position of “proxy3” within that structure. As a result, the “proxy4” and “proxy5” become incoherent within the structure and are therefore deleted. Moreover, the “proxy6” element takes the structural position of the deleted “proxy3”, and also becomes the *Real Subject* structural element for the “proxy1” and “proxy2” elements.

Figure 4.30, in turn, represents the application of the *Reduce*, in its two forms, to a *Facade*

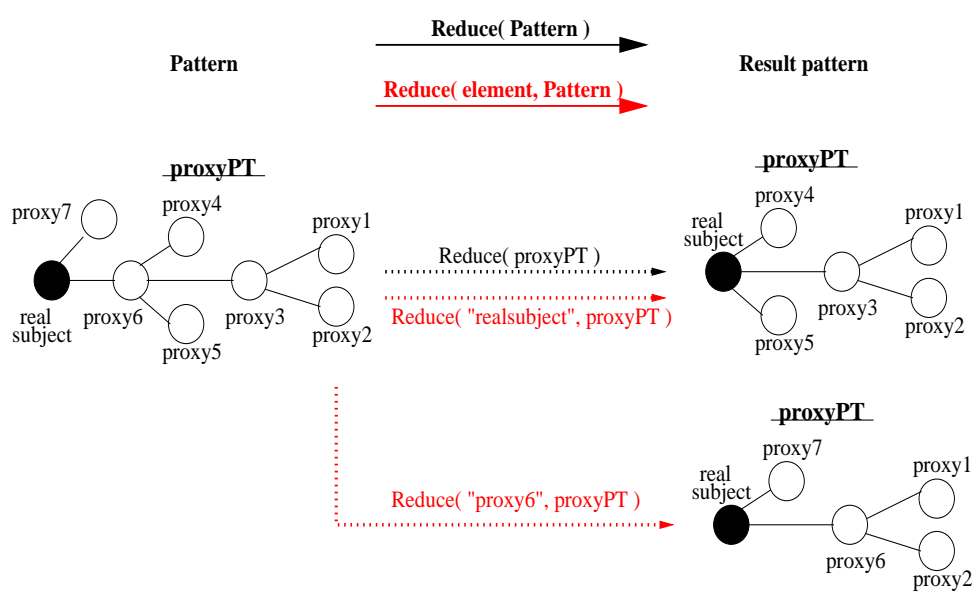


Figure 4.29: Applying the two versions of the Reduce operator to the "proxyPT" template.

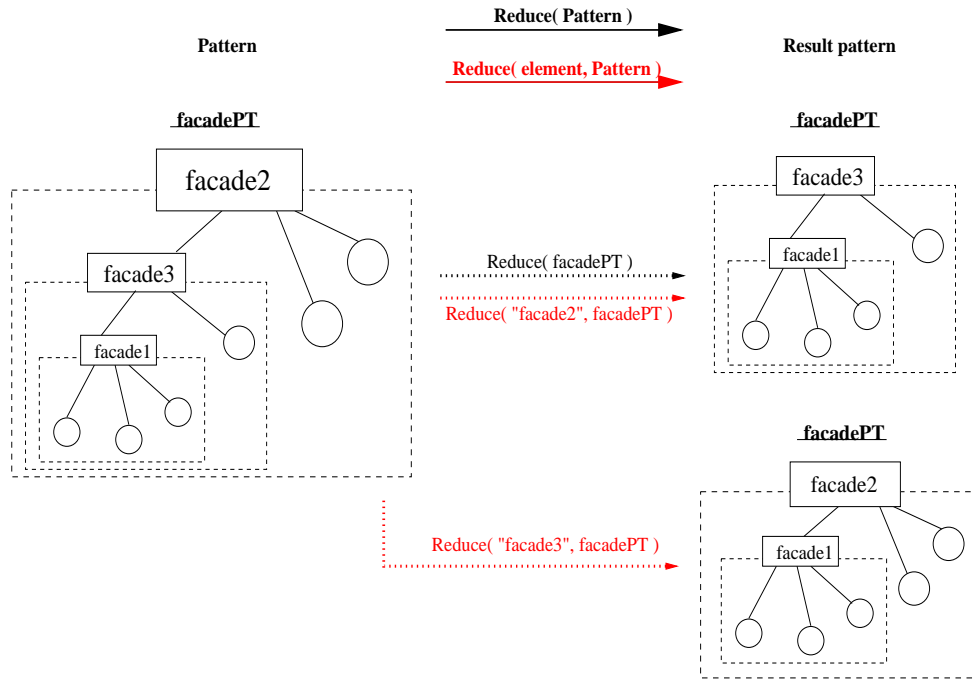


Figure 4.30: Applying the two versions of the Reduce operator to the "facadePT" template.

S-PT. The application of the $Reduce(facadePT)$ operator deflates the structure of "facadePT" by eliminating the outmost Facade structure. This result, which is pre-defined, is equal to explicitly invoking $Reduce("facade2", facadePT)$ and results in the deletion of the "facade2" element, as well as of the two associated CPHs. The $Reduce("facade3", facadePT)$ operator, in turn, eliminates the Facade structure in the middle which is represented by the "facade3" structural element. Appropriately, this CPH is eliminated along with its associated CPHs, and the structure is deflated. We recall that the $Reduce(element, facadePT)$ could only be applied to the "facade1", "facade2", and "facade3" elements within the original structure of "facadePT".

To conclude, what was said for a sequence of the *Extend*, *Increase*, and *Reduce* operators,

would also be applicable if the *Decrease* operator was used instead of the *Increase* operator. On the other hand, an operator sequence including an *Extend* and a *Reshape* is not considered in this work, although the cardinality of an extended pattern might permit a *Reshape* operation upon that pattern.

4.3.3 Structural Operation of Hierarchical Pattern Templates

This sub-section describes examples of sequences of Structural Operators that include the *Embed* or *Group* operators. For simplification reasons, these operators are considered to have been already applied, and consequently, the discussion is restricted to the structural manipulation of instances of *Hierarchical Patterns* (i.e. which were generated by *Group* or *Embed*).

Some Structural Operators are similarly applicable to all types of *Hierarchical Patterns* and with analogous results to the described application of those operators to non-hierarchical patterns. These operators are the *Eliminate*, the *Replace*, and the *Replicate*. Specifically:

1. The *Eliminate* operator results in the deletion of the outmost pattern as well as all inner patterns. Therefore, the action of the *Eliminate* operator is always recursive for any type of *Hierarchical Pattern*.
2. The *Replace* operator is also applicable to any type of *Hierarchical Pattern*, substituting it for the pattern defined as argument to the *Replace*. The *Replace* can also be applied to (inner) patterns enclosed by a *Hierarchical S-PT*, since these are directly accessible for manipulation.
3. It is possible to duplicate any kind of *Hierarchical S-PT* through the *Replicate* operator, generating new *Hierarchical S-PTs*, each one with its own identifier. Please note that each accessible individual pattern within a *Hierarchical S-PT* can also be replicated, but the replicas are generated at the same level of that *Hierarchical S-PT*. This means that the replicas of the inner patterns are created outside the *Hierarchical S-PT*, i.e. they do not become new members of that *Hierarchical S-PT*. Although keeping those new replicas as new members of a group-based *Hierarchical S-PT* would not disrupt its semantics, for a *Hierarchical S-PT* resulting from an *Embed* operation, the replicas have to be created outside this *S-PT*, in order not to disrupt its structural definition.

Likewise, the *Group*, *Ungroup*, *Embed*, and *Extract* operators can be applied to any type of *Hierarchical S-PTs* as previously described in section 4.2.2.

However, the *Reshape* operator is not applicable to any kind of *Hierarchical Pattern*. For group *S-PTs* in particular, these do not have a structural definition among the group members, so it is meaningless to reshape an inexistent structure. As for embed-based *Hierarchical S-PTs*, at least one of its component place-holders (CPH) is instantiated with another pattern, making it different from the other free CPHs. If possible, a reshape operation would require defining where to place that instantiated CPH, and this would result in different possibilities. Therefore, we discard this option for the time being.

In the following, we describe the application of the remaining operators, first to group-based hierarchical patterns, and subsequently to hierarchical patterns that resulted from the *Embed* operation.

Group-based Patterns

The *Increase*, *Decrease*, *Extend*, *Reduce*, and operators cannot be applied to a *Hierarchical S-PT* that resulted from the *Group* operator, as a group/aggregate is merely a set of structurally unrelated patterns. As described before, the structural binding between them is simply that they belong to the same group, although this group is a Structural PT on its own. This means that, for example, the *Increase* operator cannot be directly applied to a S-PT which is a group like the “groupPT” represented in Figure 4.15. However, since the members of a group are directly accessible, the *Increase* operator can be applied to each member individually, as long as these members are not groups themselves. For example, an operation like *Increase*(3, groupPT.starPT) is valid within the context of the group S-PT described in that Figure.

Patterns with Embedded Patterns

– Application of Increase/Decrease

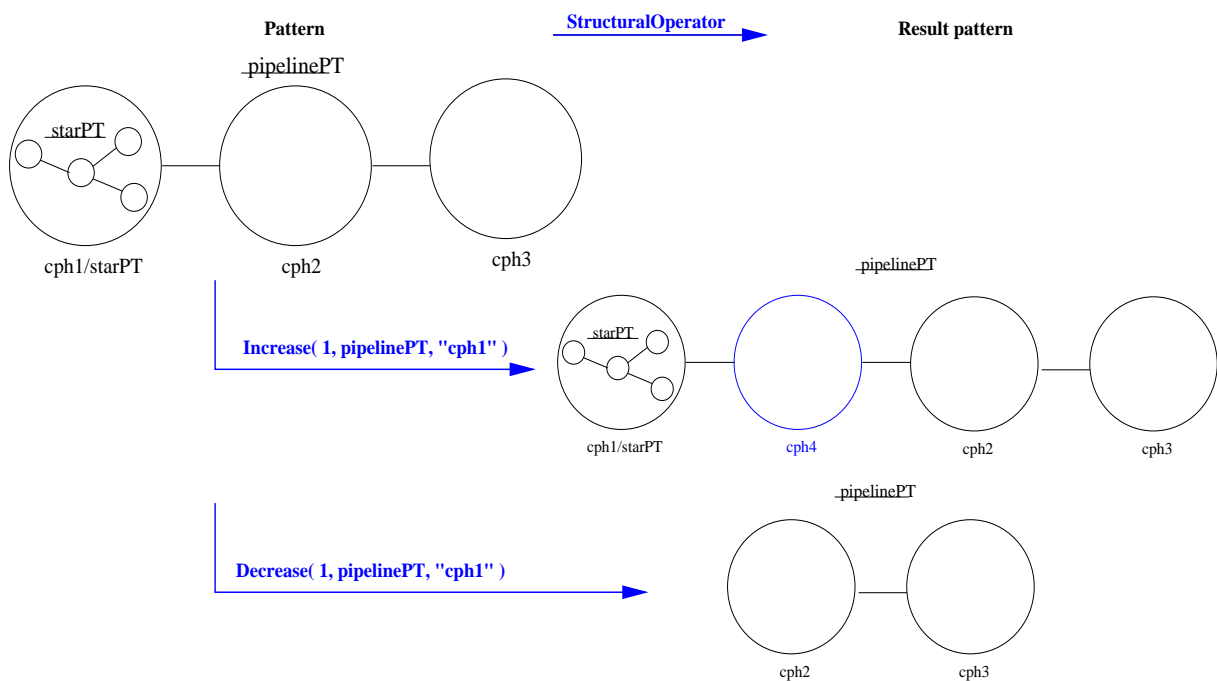


Figure 4.31: Applying the *Increase* and *Decrease* operators to a pipeline hierarchic pattern that contains an embedded pattern in the first stage.

The application of the *Increase* and *Decrease* operators to change the number of elements in a pattern with other embedded patterns is similar to what was previously described for non-hierarchic patterns. Figure 4.31 presents the case of a *Pipeline S-PT* with a *Star S-PT* embedded in its first stage. The addition of new CPHs to the “pipelinePT” can be performed either through the *Increase*(*n*, pipelinePT) operator, or in case it is necessary to define an explicit position for the new CPHs, the addition can be accomplished with the *Increase*(*n*, pipelinePT, position) operator. The Figure represents an example of this situation where the *Increase*(1, pipelinePT, “cph1”) operator generates a new component place-holder (“cph4”) which is positioned after the component place-holder “cph1”. The cardinality of “pipelinePT” becomes therefore equal to four.

The elimination of CPHs from a pattern with other embedded patterns can be accomplished through the *Decrease*(*n*, *P*, *position*) operator. Figure 4.31 presents the case of eliminating the first stage of the “pipelinePT”. This is achieved by the *Decrease*(1, *pipelinePT*, “cph1”) operator or, likewise, by the manipulation *Decrease*(1, *pipelinePT*, “starPT”) which define that one element starting at, and including, the element “cph1” (also tagged “starPT”), is to be deleted from “pipelinePT”. As a result, that first CPH in the sequence, i.e. “cph1”, is deleted along with the embedded pattern. In this case, the cardinality of “pipelinePT” becomes two.

Please note that both the *Increase* and *Decrease* operators are still directly applicable to the embedded patterns of a Hierarchical pattern. For example, the application of the *Increase*(2, *pipelinePT.starPT*) operator would increase the number of CPHs of the embedded star by two.

– Application of Extend/Reduce

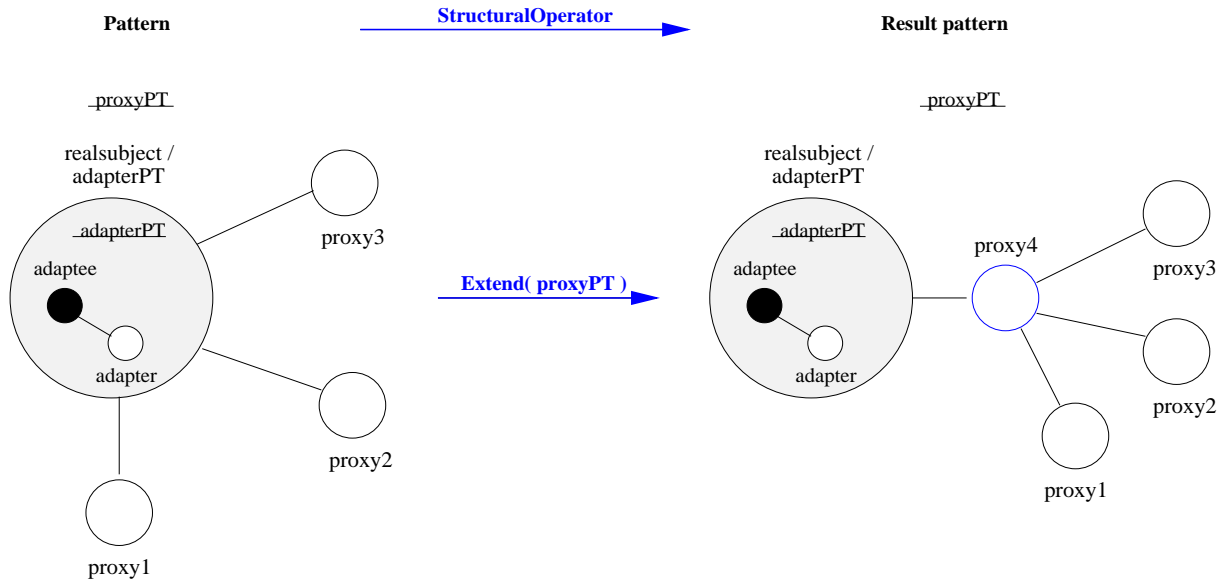


Figure 4.32: Applying the *Extend* operator to a proxy hierarchic pattern that contains an embedded pattern in the “realsubject” element.

As for the application of the *Extend* and *Reduce* operators to a pattern with other embedded patterns, it is also similar to what was previously described for non-hierarchic patterns. For example, Figure 4.32 presents the case of extending the structure of a *Proxy* S-PT, i.e. “proxyPT”, which has another pattern (“adapterPT”) embedded in the element representing the *Real Subject* within the *Proxy* pattern definition. As represented in the Figure, the application of the *Extend*(*proxyPT*) operator results in the creation of the element “proxy4” that acts as surrogate to the pre-existent proxies (i.e. “proxy1”, “proxy2”, and “proxy3”). Moreover, the “adapterPT” pattern remains embedded in the “realsubject” element within the structure. Clearly, the structure of “adapterPT” could also be augmented by direct manipulation through the *Extend* operator.

Figure 4.33, in turn, presents a case of reducing the structure of a facade hierarchic pattern. Specifically, the “facadePT” has the pattern “pipelinePT” pattern embedded in the outmost facade element, i.e. “facade2”. The application of the *Reduce*(*facadePT*) operator defines that the outmost facade element is deleted. Consequently, the “facade2” element is deleted along with the embedded “pipelinePT”.

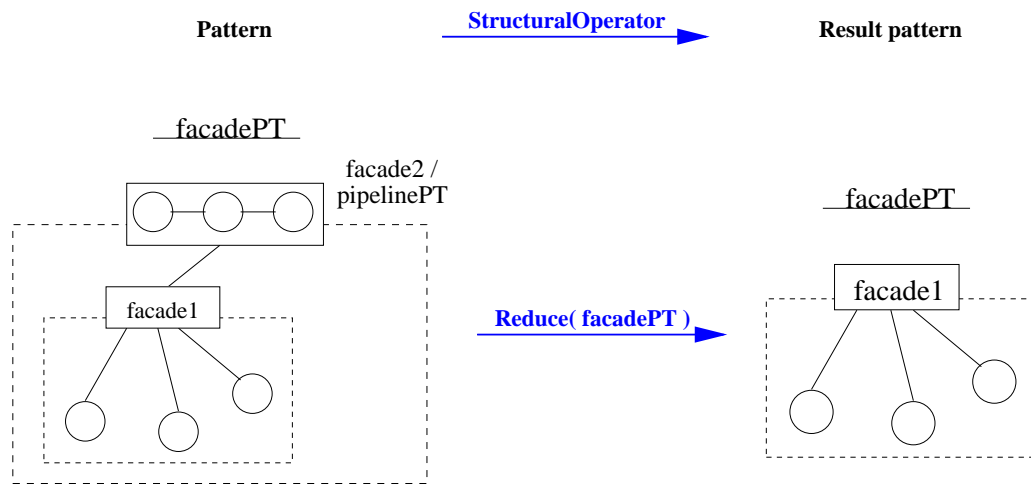


Figure 4.33: Applying the Reduce operators to a facade hierarchic pattern that contains an embedded pattern in the outmost facade element (i.e. “facade2”).

4.4 Semantics of Behavioural Operators

Despite different types of *Behavioural Operators* were introduced in section 3.3, the discussion in this sub-section is restricted to the semantics of the *Execution Operators*. These enable direct control of ongoing computations, as it will be illustrated in the applications of Chapter 7. Although the available *Execution Operators* were previously described (section 3.3.5) in terms of pattern templates, in fact, those operators act upon *Pattern Instances (PIs)*. As explained before, PI are Structural Pattern Templates which have been already assigned to one ore more *Behavioural Patterns*, and within which component place-holders have already been bound to executable component instances.

Execution Operators include: *Start* and *Terminate*, *Stop* and *Resume*, *Restart* and *TerminateRestart*, *Limit*, *Repeat* and *TerminateRepeat*, *Log* and *TerminateLog*, *SeqLog* and *TermSeqLog*, and *ResumeLog*, *Steer*, and *SteerComponent*. The latter two operators were defined as being implementation dependent and so their semantics is not discussed. As for all the other *Execution Operators*, their semantic discussion is based on specific examples. Concretely, all operators are considered to be applied to the same *Pattern Instance (PI)* – a three-stage pipeline coordinated by the *Streaming Behavioural Pattern*.

The explanation of the *Execution Operators*’ semantics is preceded by a sub-section describing the main concepts of the used formalism, namely the CO_OPN/2 [32,33] synchronous model. Albeit this formalism provides also asynchronous operations (for example to mimic an arbitrary selection of the invocation of one of two available methods), all the CO_OPN/2 operations we use are synchronous ones. For example, the application of some *Behavioural Operators* to a pattern instance (e.g. *Stop* and *Resume*) results on a synchronous call to several methods available at the elements belonging to that pattern instance. Even though a complete semantics discussion should also consider cases where the invocation of each component is asynchronous, we restrict our analysis to the synchronous case, primarily because of the particular benefit the CO_OPN/2 tool offers in undertaking such analysis. Specifically, our goal is simply to present a clear definition of the workings of each operator, although this does

not imply that the actual implementations of the described operators should follow a strictly synchronous model.

4.4.1 The CO_OPN/2 Formalism

The *CO_OPN/2* formalism [32, 33] provides Object-Oriented abstractions for specifying systems, e.g. classes and objects as class instances, and where synchronisation between object invocations can be modelled.

The functionality of each *object* within the CO_OPN/2 formalism is represented as a *Petri-Net* and data flow is based on abstract data types. The interface of CO_OPN/2 objects provide *methods* (i.e. *input ports*) with the guarantee of transactional semantics upon invocation. Moreover, CO_OPN/2 objects may generate events through its interface's *gates* (i.e. *output ports*). Considering our operators' semantics, CO_OPN/2 objects are used by us to describe both the executable components representing the pattern instances' elements, as well as other necessary functionality.

The *Context* concept within *CO_OPN/2*, in turn, defines a coordination environment for ruling inter-object interactions. Namely, the interactions between a pattern instance's elements are defined within a particular context.

In the following, we describe the above concepts and their visual notation through a small example.

I – Objects

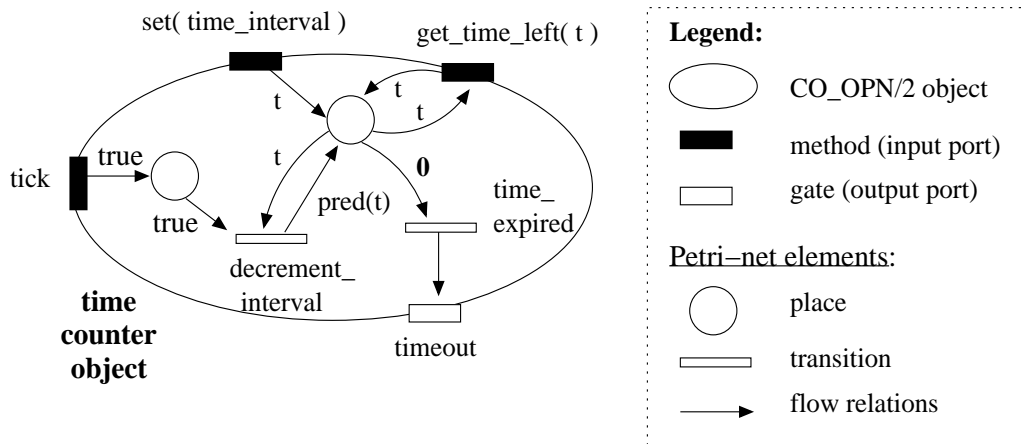


Figure 4.34: An example of a CO_OPN/2 object with its behaviour modelled through a Petri-Net.

Figure 4.34 shows a simple `CO_OPN/2` object (named *time counter*) representing a timed alarm. Based on the `CO_OPN/2` visual notation, an object is represented as an ellipse, and its interface's *input* and *output ports* are represented as solid small rectangles at the border of the ellipse, black solid or white solid, respectively. The object in Figure 4.34 provides three invocable methods (i.e. *input ports*), namely, *tick*, *set(time_interval)*, and *get_time_left(t)*. A method's name is depicted next to its correspondent small black solid rectangle at the border

of the ellipse, and its invocation is to be represented by a arrow-head (dashed) line directed to that black rectangle, as will be described further ahead. The *tick* method, in particular, is to be invoked by another object representing a clock, whereas the *set* method is to be used to define the time interval of the alarm, and the *get_time_left* method allows inspecting the remaining time left before the alarm is fired. Upon expiring of the time interval, the *time counter* object generates an event through an *output port* or *gate* named *timeout*. Therefore, *Gates* in *CO_OPN/2* represent events generated by the object and they are represented as small white solid rectangles along the ellipse's border.

Being a *CO_OPN/2* object, the timed alarm's functionality is modelled through a *Petri-Net* inside the object. The *Petri-Net* elements are defined according to the following notation: a) *places* are represented as small white circles; b) *transitions* are modelled as small white rectangles; c) *input arcs* (connecting places to transitions) are symbolised as arrow-headed arcs from places to transitions; and d) *output arcs* (connecting transitions to places) correspond to arrow-headed arcs from transitions to places. Transitions are fired as soon as all required tokens (existent in places connected with input arcs) are present. As mentioned before, tokens in *CO_OPN/2* are Abstract Data Types (ADTs) allowing the definition of simpler *Petri-Nets* to model an object's functionality.

The *time counter* object contains two places, and two transitions labeled "decrement_interval" and "time_expired". One place receives the time interval which is represented by the "t" token, and it is initialised through the invocation of the *set* method. The other place receives a token upon each tick of a clock, which results from the invocation of the *tick* method. The "decrement_interval" transition is fired as soon as each tick token is ready at one place, and the "t" token (with "t" bigger than zero) is ready at the other place. As a result, the value of the time interval (i.e. the "t" ADT) is decremented and it is set again at the same place. Only, when the value of the token "t" becomes zero, the "time_expired" transition is fired consuming the token and generating an event through the *timeout* gate. Meanwhile, the remaining time before the timeout event can be inspected by invoking the *get_time_left(t)* method. In this case, the value of the token "t" is returned as an output parameter. As such, upon one method invocation in *CO_OPN/2*, the flow of data may be bi-directional, as a result of the presence of both input and output parameters in that object's method.

II – Inter-object Synchronised Calls

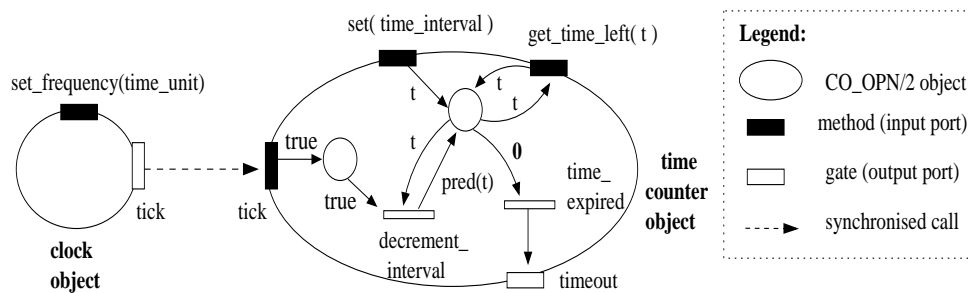


Figure 4.35: An example of a synchronised call between two *CO_OPN/2* objects.

implemented within that context), whereas output ports symbolise the context's events or services required by that context. According to the formalism, the context's ports are represented as bi-coloured rectangles. On one hand, inport ports are represented with the dark part of the rectangle on the outside of the context and the white part on the inside. On the other hand, output ports are represented with the dark side of the rectangle inside the context box and the white side is on the outside of that box. Figure 4.36 shows that the *Alarm context* provides the *SetAlarm* and *TimeLeft* methods, and the *alarm* output port.

In *CO_OPN/2*, the calls among contexts, and also between a context and its inner objects or between the context and its sub-contexts, consist of synchronised invocations from output ports to input ports where multiple invocations may be ruled by *synchronisation policies*. *CO_OPN/2* provides three kinds of *synchronisation policies* where, similarly to transactions, all the policies imply an "all or nothing" semantics. First, the *simultaneity* policy implies a *simultaneous invocation* of the involved calls (and one call can only succeed if all other involved calls can succeed as well). Second, the *sequence* synchronisation policy defines a sequential call for the involved methods (e.g. one before the other, in a sequence policy applied to two synchronised calls). Finally, an *alternative or nondeterminism* policy defines that the method to be executed is selected in a non-deterministic way among a set of available alternatives (i.e. the call succeeds if one of the alternatives succeeds).

In the example depicted in Figure 4.36, and as a result of the *Simultaneity* policy, an invocation of the *SetAlarm(time_unit, time_interval)* context's method implies a simultaneous synchronous call to the *set_frequency(time_unit)* and *set(time_interval)* methods at the *clock* and *time counter* objects, respectively. This guarantees that the initialisation of the *clock* object with the chosen frequency (e.g. seconds, milliseconds) is synchronised with the initialisation of the alarm (*time counter* object) with the expiration time interval (defined according to the chosen frequency). When the time expires, an event at the *timeout* gate at the *time counter* object generates an event available at the *alarm* output port of the *Alarm context*. This context also provides the *TimeLeft* method to inspect meanwhile the interval of time until the next alarm.

The next-subsections describe the semantics of the *Execution Behavioural* operators according to the *CO_OPN/2* formalism. Implementation-wise, to note that those descriptions rely on the assumption that each executable component within a pattern, e.g. that represents a particular tool or service, includes a *wrapper*. This wrapper is responsible for implementing the interface to the component and providing proper coordination and communication to the necessary resource managers (that support that tool or service execution). Furthermore, the operators may have effect only over those wrappers. This means that if those wrappers are interfacing the access to, for example, services over which it is not possible to have a direct execution control, the execution control represented by the described operators are restricted to the manipulated wrappers. For instance, whereas in some cases it may be possible to suspend the execution of the executables forming the stages of a *Pipeline Pattern Instance (PI)*, in some other cases the suspension may be restricted to the execution of the wrappers interfacing (some of) the *Pipeline PI's* stages to specific services/tools.

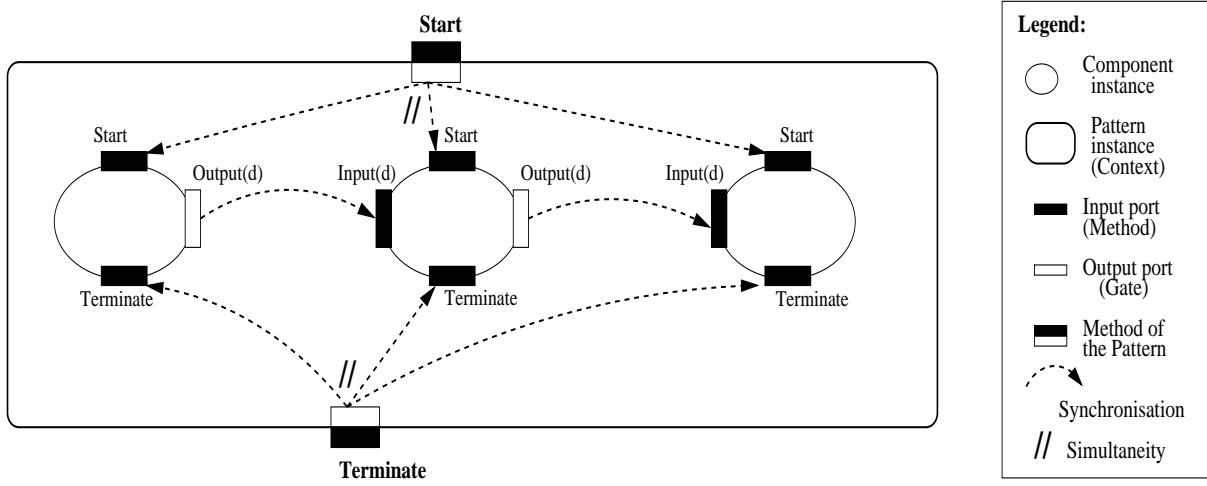


Figure 4.37: Example of the *Start* and *Terminate* operators over a pipeline pattern instance.

4.4.2 Start and Terminate Operators

Figure 4.37 presents an application example of the *Start* and *Terminate* operators over the selected pattern instance, namely a pipeline with three stages. Each stage represents an executable component (e.g. tool/service) which is modelled as an *CO_OPN/2 object* (i.e. a small ellipse). As we are not, in this discussion, concerned with the internal behaviour of such components, we do not show Petri-Net blocks for these. As described before, the input ports (i.e. *method calls*) available on a component are represented by black rectangles along the border of the ellipse, whereas the output ports (i.e. *Gates*) are represented as white rectangles.

Figure 4.37 shows that the output port (*Output(d)*) of the leftmost component in the pipeline instance is synchronised with one of the input ports (*Input(d)*) of the middle component. As soon as the *Output* method is invoked, the *Input* method in the other component is invoked as well, and data is exchanged in the process by argument instantiation. Likewise, the output port (*Output(d)*) of the middle component is synchronised with the port *Input(d)* of the rightmost component. In this way we represent, in a simplified way, the *Streaming Behavioural Pattern* ruling the control and data flows between the components of the pipeline.

A *CO_OPN/2 context* is used to represent the pipeline Pattern Instance, as well as the necessary synchronised calls to the pattern as a whole. The invocation of the *Start* method over the pipeline context implies the *simultaneous invocation* of the *Start* method of every component. We represent this simultaneous invocation by the *simultaneity policy* symbol “//”, according to the *CO_OPN/2* formalism. The *Terminate* operator has a similar behaviour to the *Start* operator, as shown in the Figure. The invocation of *Terminate* over the pipeline implies the simultaneous invocation of the *Terminate* method at all component instances. Although omitted from the context, an extra *CO_OPN/2 object* would represent the pattern instance’s state, namely, if the pipeline as a whole is executing or has terminated (i.e. it is not running). Nevertheless, it is assumed that a call to the *Terminate* operator is only effective if the pattern is already running.

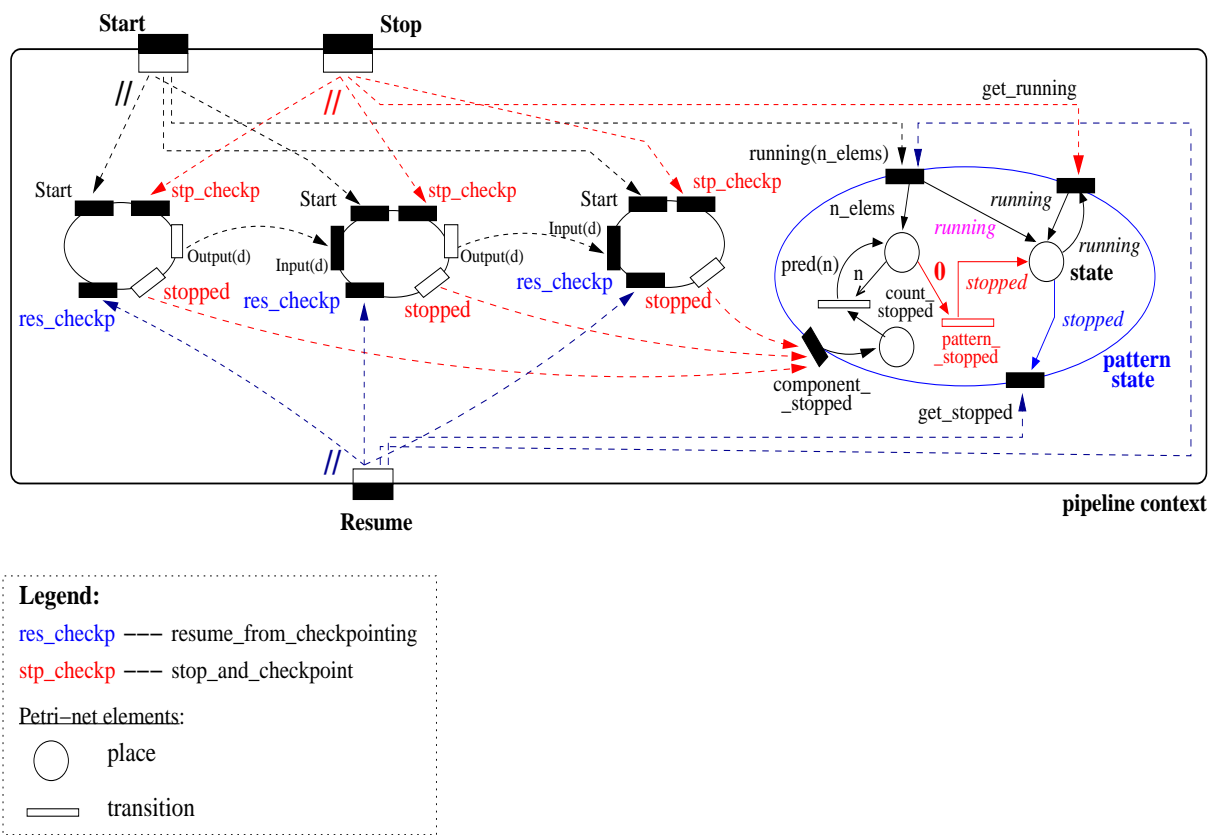


Figure 4.38: Example of the Stop and Resume operators applied to a pipeline instance.

4.4.3 Stop and Resume Operators

The semantics of the *Stop* operator implies the suspension of the execution of all component instances – hence it is similar to the *Terminate* operator. Using this operator, however, also causes the state of all component instances to be recorded. Each component (in fact, the component wrapper) is assumed to have a *stp_checkp* method that upon being called suspends (or requests the suspension of) the inner component's execution and saves its current state. As consequence of that method call, an event is generated through the *stopped* gate (output port) informing that the component is stopped/suspended.

Figure 4.38 represents the semantics of *Stop* operator applied to the pipeline instance as a simultaneity synchronous call to all of *stp_checkp* methods provided by the components. An extra *CO_OPN/2* object, included in the pipeline context, represents the pattern's state. This *pattern state* object registers whether the component is running or stopped (a possible terminated/not_running state is omitted for simplification reasons). As shown in the Figure, the *Start* operator invokes the *running* method and the components' individual *start* methods (with a *simultaneity policy*), and in consequence, the pattern is set to be in the "running" state and the number of the pipeline's stages is defined (through the "n_elements" token). Although not represented in the Figure, the number of elements (i.e. stages) of the pipeline is assumed to be available at the pattern (i.e. pattern instance) context. This value instantiates the argument "n_elements" when the method "running(n_elements)" is invoked as a consequence of the *Start* operator. From now on, we assume that the information about the number of elements forming

a pattern instance is always available to be used, and it is updated automatically whenever that number changes (e.g. as a result of applying the *Increase/Decrease Structural Operators*).

The *Stop* operator, in turn, only succeeds if the pattern is running, as this operator requires a successful call to the *get_running* method (which requires a “running” token at the place “state”). Upon a successful stopping operation, and after all *stopped* events get collected through calls to the *component_stopped* method, the pattern instance’s state switches to “stopped”.

The *Resume* operator consists on a synchronised call to the individual *res_checkp* methods and to the *get_stopped* and *running* methods of the pattern state object. Therefore, the effectiveness of the operator depends also on the pattern being in the “stopped” state, and the latter is then changed to “running”. The invocation of each component’s *res_checkp* method implies restoring that component’s saved state and resume its execution.

4.4.4 Repeat and TerminateRepeat Operators

The *Repeat(n,P)* operator guarantees that the pattern instance “P” is executed “n” times, where the next iteration is started as soon as the previous execution ends. In case the user wants to abort those consecutive executions, the *TerminateRepeat(P)* operator aborts the current execution and prevents the execution of the remaining iterations.

As depicted in Figure 4.39, the semantics of *Repeat* operator relies on two contexts: the *Repeat* and the *pipeline* contexts. The former context contains a *CO_OPN/2* object which is responsible for launching the pattern’s first execution, as well as the subsequent ones as soon as a previous execution is detected as ended. The object contains the *remaining_iterations* counter which is decremented when the *next_iteration* context’s method is called, and can also be set to zero through the *terminate_repeat* method. A zero value for the Petri-Net place *remaining_iterations* fires a transition triggering the end of the repeat.

The *pipeline context* represents the pattern instance’s components and their interactions, and includes also an object that controls when to request for the next pattern’s execution. The latter object also registers whether the pattern is in the “repeating” state (represented by “rept” in *pipeline state object* in the Figure) or in the “terminated” state. The former state is set by the *define_first_iteration* context method, and the latter is set by the *TerminateRepeat* operator. This operator is associated (as a method) to the pipeline context for simplification reasons.

The *Repeat* operator implies consecutive calls to the *Start* method at the *pipeline context*, and each call results in the invocation of each component’s *start* method (with a *simultaneity policy*), as well as of the *started* method belonging to the *pipeline state* object. As a result of invoking this *started(n_elems)* method, the number of elements (i.e. components) within the pattern instance’s structure (three in this case) is saved in a counter (a place within the *pipeline state* object) that is used to represent how many pipeline elements have not terminated executing yet. The value of that counter is initialised with the value of the “n_elems” argument that is instantiated when the *started* method of the *pipeline state* object is invoked. As said before, it is assumed that the number of elements forming a pattern instance’s structure is known within the context that represents the instance, namely, *pipeline context* in this case.

Whenever the execution of one pattern element (i.e. component) ends, the value of the

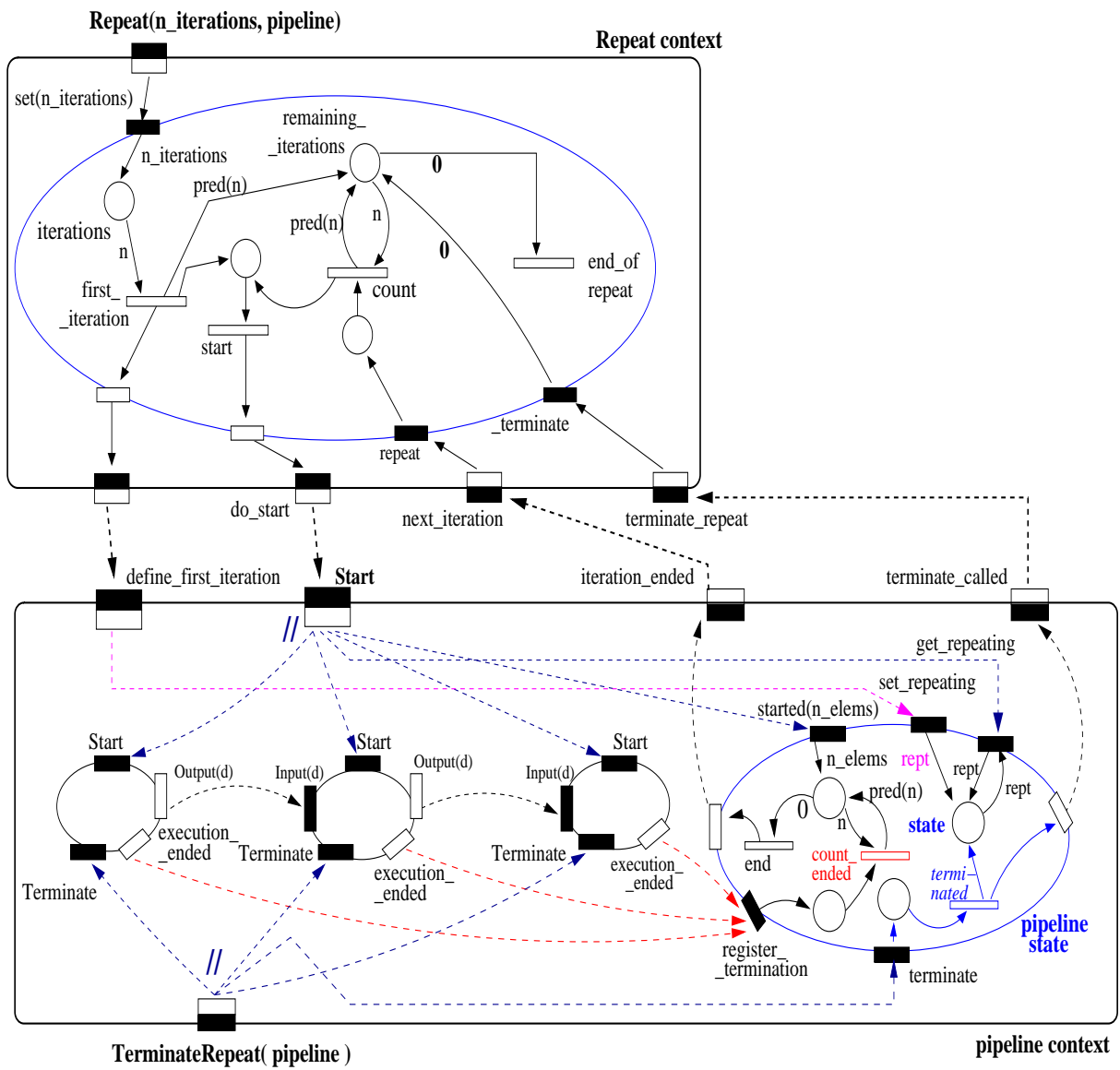


Figure 4.39: Example of the Repeat and TerminateRepeat operators applied to a pipeline instance.

above counter is decremented. Such happens when each component (in fact, each wrapper) generates an event through its *execution_ended* gate upon executing ending. As a direct consequence of this, the *register_termination* method in the *pipeline state* object is invoked. When the counter reaches zero, that object generates an event that results on a call request to the *next_iteration* method available at the *Repeat* context.

Finally, the *TerminateRepeat* operator implies a synchronisation call ruled by the *simultaneity* policy to the *Terminate* method at each pipeline element, as well as to the *terminate* method of the *pipeline state* object. In this object, the transition “terminated” sets the “state” place with a “terminated” token. On one hand, this denies an immediate re-start, i.e. at this time an invocation to the *Start* method fails since the *get_repeating* method in the *pipeline state* object cannot succeed because the token at the place “state” is not “rept”. On the other hand, the transition “terminated” also generates an event through one gate of the *pipeline state* object, which in turn will result on the invocation of the *terminate_repeat* method at the *Repeat* context.

This method, in turn, invokes the `_terminate` method that sets to zero the value of the place “remaining_iterations”. Consequently, the sequential re-start of the Pattern Instance triggered by the *Repeat* operator is interrupted.

4.4.5 Limit Operator

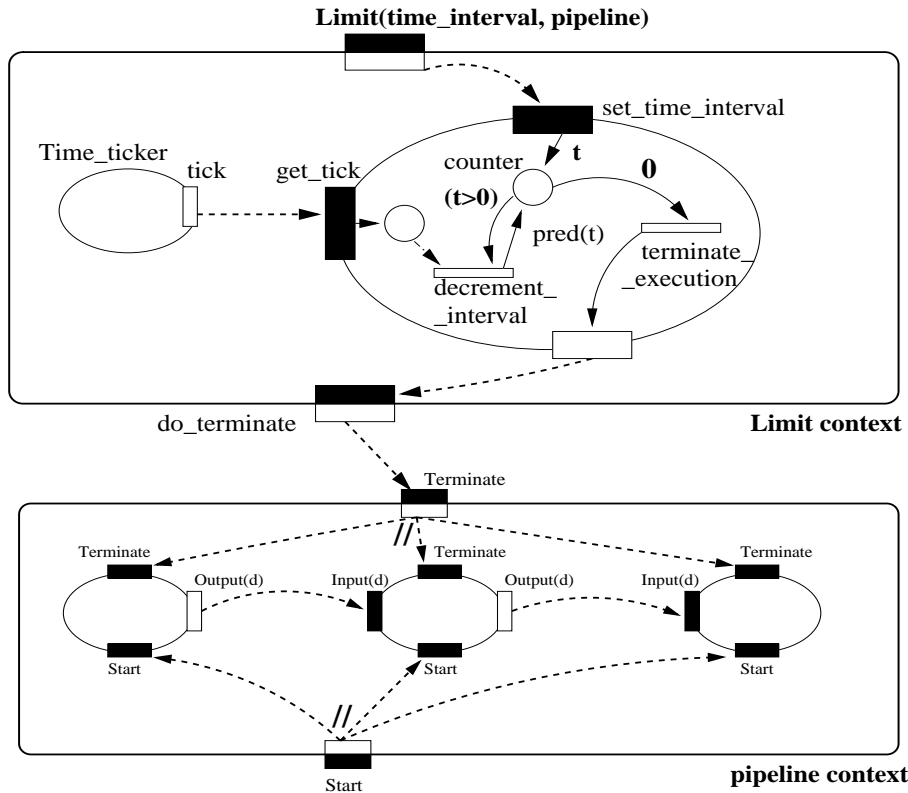


Figure 4.40: Example of the *Limit* operator applied to a pipeline instance.

The $Limit(\delta T, P)$ operator waits until the time value received as input (i.e. δT) expires – followed by the termination of the Pattern Instance managed by this operator, in case the pattern is still being run. It is assumed that the *Limit* operator is applied to a running pattern instance (i.e it may be composed with or applied in sequence after the *Start* operator), and its action is restricted to that particular execution. This means that in order to limit the time of the pattern’s next execution, it is necessary to apply the *Limit* operator again.

The description of the *Limit* operator’s semantics includes two contexts: the *Limit* and the *pipeline* contexts, as represented in Figure 4.40. The *Limit* context consists of an object that registers the time interval, decrements this interval at each tick of a clock object, and when it reaches the zero value a transition is fired prompting the operation of the *Terminate* method on the pipeline Pattern Instance. As depicted in the Figure, a zero value for the *time_interval* parameter of the *Limit* operator originates an immediate call to the *Terminate* method in the *pipeline* context.

The *Terminate* method, in turn, causes a synchronisation call with a *simultaneity policy* to the *terminate* methods available at the components’ interfaces. For simplification reasons, the pattern instance’s state is omitted although it is also assumed that the effectiveness of the *Terminate*

method depends on the pattern being in the running state.

4.4.6 Restart and TerminateRestart Operators

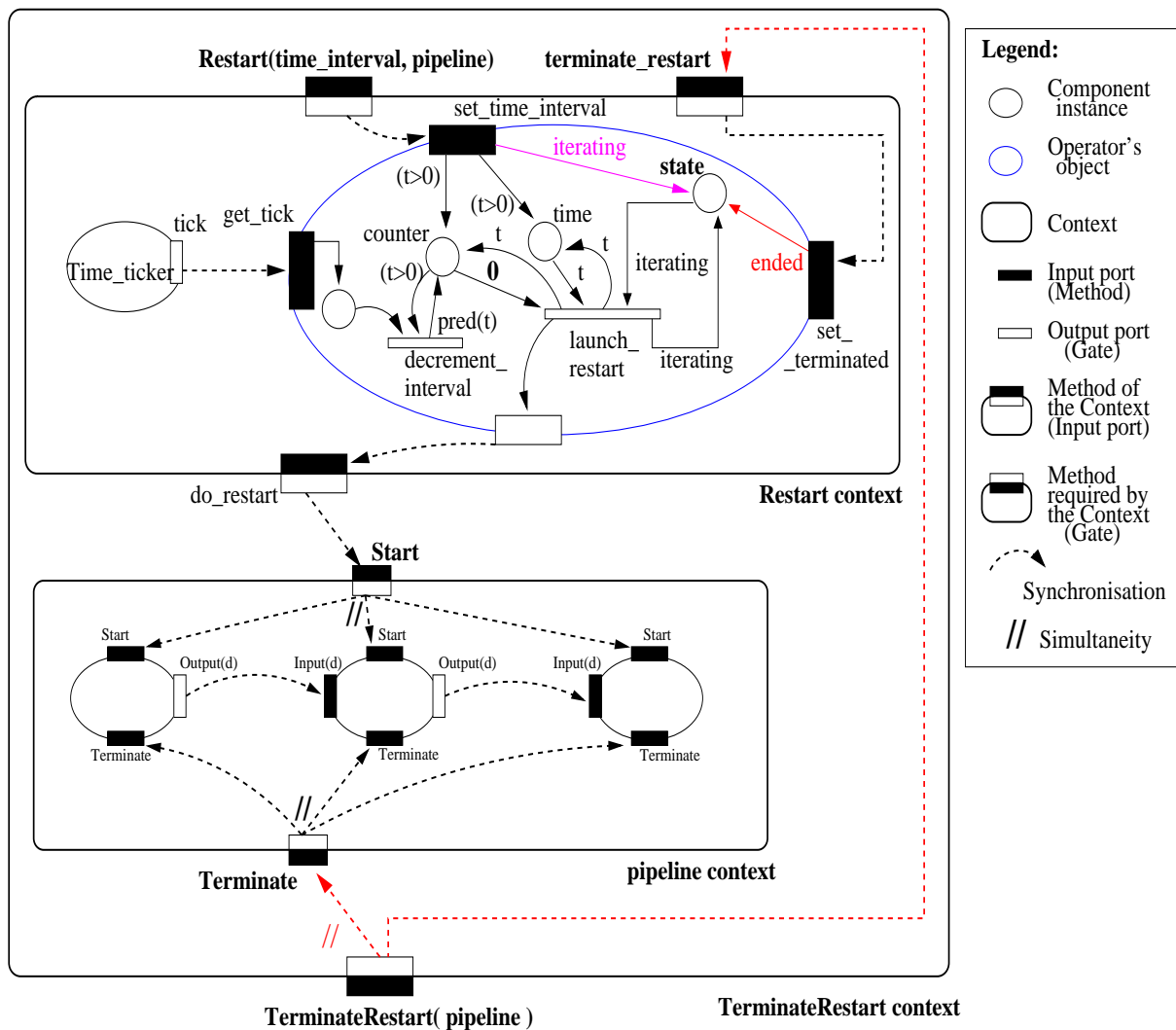


Figure 4.41: Example of the `Restart` and `TerminateRestart` operators applied to a pipeline instance.

The *Restart*($\delta T, P$) operator is similar to the *Limit* operator described previously, in the sense that both operators depend on the notion of time for controlling a pattern. Namely, when the time value received as input (i.e. δT) expires, an operation is performed over the Pattern Instance represented by the parameter “P”. However, the *Restart* operator results on the invocation of the *Start* operation over the Pattern Instance it manages, after waiting for the expiration of the value held by the input time token. Moreover, the *Restart* operator guarantees a periodic re-launching of the pattern it is applied to – the time token is saved and after launching one particular execution the next execution will start as soon the received time interval expires again.

The *TerminateRestart(P)* operator, in turn, discontinues the periodic re-start set by the *Restart* operator and terminates a possible current execution. The existence of the *TerminateRestart*

operator separated from the *Terminate* operator, allows the division between: a) the termination of a single (current) execution (which is done with the *Terminate* operator) and that not prevents the subsequent periodic re-starts; and b) the interruption of the periodic re-starts, guaranteed by the *TerminateRestart* operator.

The semantics of the *Restart* and *TerminateRestart* operators are depicted in Figure 4.41. The semantic description of the *Restart* operator relies on two contexts, the *Restart context* and the Pattern Instance context it is applied to (in this case, the *pipeline context*). Since the semantics of the *TerminateRestart* depends on the two previous contexts, these are represented as sub-contexts of the *TerminateRestart* context, allowing a simultaneity synchronisation call to the pipeline context's *Terminate* method and to the *terminate_restart* method of the *Restart* context. This latter method changes the state of the present *Restart* operation to *ended*, preventing further re-starts.

The *Restart* context encapsulates two *CO_OPN/2* objects: one is a timer which generates a tick at a specific time interval (e.g. a second); the second object represents the necessary steps for the restart operator. One of the transitions in the Petri-Net in this second object decrements the time interval (received as argument) at each tick of the timer, and keeps the result in the *counter* place. When the *counter* reaches zero, a second transition is fired which launches the restart of the pipeline's execution through the *do_restart* gate, and also re-initialises the place "counter" with the original time interval (kept in the place "time"). The call of the *Restart* operator also initialises the place "state" defining that the process is "iterating". The firing of the *launch_restart* transition also depends on the value of that place. In fact, if the *TerminateRestart* operator is called meanwhile, the *state*'s value changes to "ended", preventing the automatic restart. Upon the next call to *Restart*, the *state*'s value is set again to "iterating".

As can be concluded from Figure 4.41, the semantics of the *Restart* operator is disassociated from the time that the Pattern Instance, which the operator is applied to, takes to complete its execution. Therefore, it is assumed that the user defines a reasonable *time_interval* for re-launching the execution (i.e. the value of the δT parameter should be greater than the Pattern Instance's execution time). For example, for some applications it may be useful to launch their execution on a weekly basis, whereas for others some small *time_intervals* are due (e.g. frequent processing of sensor based collected data which, typically, require application dependent *time_intervals*). The *Restart* operator may guarantee a periodic execution of an individual pattern configuration that only supports part of an application's (pattern-based) configuration, and independently from the overall configuration's execution being ruled by other Behavioural Pattern Operators.

4.4.7 Log Related Operators

Logging operators allow the user to checkpoint the state of a running pattern instance, either (1) once, (2) periodically, or (3) sequentially, and (4) to re-start the pattern's execution from one of those saved checkpoints. Specifically, the *Log* operator supports the first two types of log operations (1 and 2), whereas the *TerminateLog* operator discontinues the periodic logging defined by operation (2), and the *ResumeLog* enacts a pattern (re-)execution from a saved checkpoint (operation 4). Finally, the *SeqLog* operator executes a consecutive checkpointing (operation 3) that

can be discontinued by the *TermSeqLog* operator. It is assumed that the *Log* operators (except for the *ResumeLog*) are only applied to an already executing *Pattern Instance*.

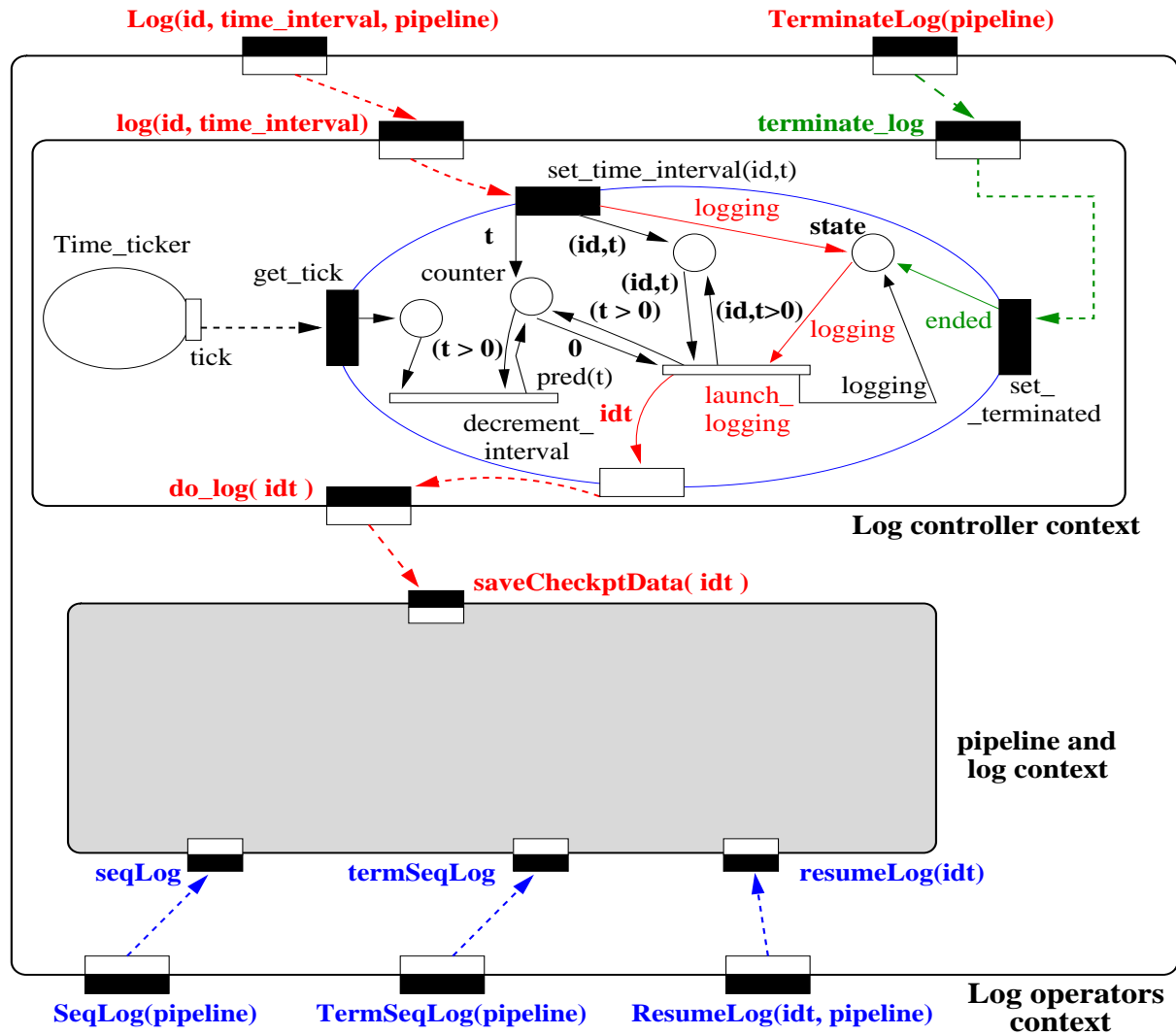


Figure 4.42: Example of the *Log*, *TerminateLog*, *SeqLog*, *TermSeqLog*, and *ResumeLog* operators applied to a pipeline instance.

Figure 4.42 represents a simplified semantics of all log related operators, which are described in the following.

SeqLog and TermSeqLog

The *SeqLog*(*P*) operator launches an activity for the sequential logging of the execution of pattern “P”. This operator assumes that the PI that instantiates “P”, e.g. “pipeline” as in the *SeqLog*(*pipeline*) context method in Figure 4.42, has an associated specification of a collection of checkpoint code locations, corresponding to relevant points such that the state of the pattern should be logged when execution control reaches those points. The activation of the *SeqLog* operator therefore triggers an automatic mechanism for collecting a succession of intermediate saved global states of “P”, each one tagged with a unique identifier. Such succession constitutes a trace of the pattern’s execution and allows its off-line inspection by appropriate tools. Please

note that, as “P” will typically include multiple components which are subject to a distributed execution, the building of the trace of “P” requires the invocation of an appropriate algorithm constructing the global state checkpointing of the distributed computation. However, at the level of our pattern operator model, the definition of the *SeqLog* operator is kept independent of the particular global checkpointing algorithm used for the above purpose. This semantics is to be guaranteed within the *pipeline and log context* displayed in Figure 4.42, as a result of the invocation of its *seqLog* context method.

In fact, the *pipeline and log context* in the Figure represents the encapsulation of: a) the three stage pipeline *Pattern Instance (PI)* example used so far whose components are subject to a distributed execution; b) an additional executable entity running the appropriate algorithm for obtaining the mentioned intermediate global states of “P”; and c) a log component to save these states. Each global state is to be appropriately tagged with an identifier which can later on be used, e.g. as argument to the *ResumeLog(idt, P)* operator described ahead.

The *TermSeqLog(P)* operator, in turn, interrupts an ongoing sequential logging triggered by the *SeqLog(P)* operator.

Log and TerminateLog

The semantics of the *Log(id, δT , P)* and *TerminateLog(P)* operators are associated, as represented in Figure 4.42 for the pipeline PI. The *Log* operator may be called to perform either a single checkpointing or a periodic one, being the latter terminated by the *TerminateLog* operator. Specifically:

1. In the case of a single checkpointing, the user labels the log through the “id” parameter of the *Log* operator, and defines the time interval δT as zero. Consequently, the *launch_logging* transition is promptly fired, as represented in the object within the *Log controller context* in Figure 4.42. As a result, an event is generated at this context which is synchronised with a call to the *saveCheckptData(idt)* method at the *pipeline and log context* in the Figure. Such method represents the request for a consistent checkpointing of the global state of a *pipeline Pattern Instance (PI)*, with a similar semantics to what was described for the *SeqLog* operator. Consequently, the PI’s global state is appropriately tagged with the value passed to the “idt” parameter in the *saveCheckptData(idt)* method and is saved in the log component within the *pipeline and log context*.
2. In the case of a periodic checkpointing, the user calls the *Log(id, δT , P)* operator with a value greater than zero for the time interval (δT) parameter, and labels the logging operation with the “id” tag. As can be seen in the *Log controller context* in Figure 4.42, the “time interval” value is used to set a timer that upon expiring fires the *launch_logging* transition. Before that, the state of the log is set to “logging”, and the time interval is saved to set the timer for the next call. The *launch_logging* transition results in a request for the next logging operation through *do_log(idt)* event where the “idt” argument tags that specific log call. Specifically, the “idt” identifier is the result of the “id” tag from the *Log* operator combined with a time reference. Consequently, a call to the *saveCheckptData(idt)* method at the *pipeline and log context* is generated with a similar semantics to what was previously described. Each

individual checkpointing of the PI's global state is thereafter accessible at the *pipeline and log context* based on the "idt" identifier.

3. To finalise the periodic checkpointing, the user may call the *TerminateLog(P)* operator generating a call to the *terminate_log* method of the *Log controller context*, as represented in Figure 4.42. In consequence, the value of the "state" place changes to "ended" which inhibits the firing of the *launch_logging* transition. A posterior call to the *Log* operator will reset that place's state to "logging" allowing another periodic logging.

ResumeLog

In order to replay the execution of a pattern instance "P" starting from a previously saved state, the user may use the *ResumeLog(idt, P)* operator. The first argument, i.e. "idt", identifies a specific saved state in time of the execution of "P" which may have been produced by either the *Log* or *SeqLog* operators. As represented in Figure 4.42 for the pipeline PI example, a call to the *ResumeLog(idt, pipeline)* operator implies the invocation of a method of the *pipeline and log context* which is responsible for resuming the pipeline's execution from the global state identified by the value of "idt".

To conclude the discussion on the *Execution Operators*, we restate that the described semantics were defined in the context of a simple *Pattern Instance* example. A more complete discussion would require, for example, the semantic definition of applying the described execution operators upon *Hierarchical Pattern Instances* which may present different ruling Behavioural Patterns.

Next section provides some examples on how some *Execution Operators* may be combined, although still applied to the same non-hierarchical pipeline PI example.

4.5 Sequences of Behavioural Operators

In this section, we first define some relevant sequences of Behavioural Operators as well as possible compositions of those operators, and subsequently discuss some particular situations in the context of the *Restart* and *Repeat* operators.

4.5.1 Common Sequences and Compound Operators

One common sequential application of Behavioural Operators involves the *Start* and *Limit*, as the latter requires the operated pattern instance to be already under execution. As such, the user launches the pattern instance's execution and later may define a time limit for that execution:

```
Start( pipeline )
. . .
Limit( time_interval, pipeline )
```

In case of an immediate operation of *Limit* to a new execution of a Pattern instance, the user may also compose both operators forming a *Compound Behavioural Operator* such as:

```
Limit( time_interval, Start( pipeline ) )
```

According to what was said before, the inverse composition, namely *Start(Limit(...)*), is not allowed. Likewise, a sequence consisting of applying the *Limit* operator followed by *Start* results in the *Limit*'s action to be ignored.

Another typical Behavioural Operator sequence consists of the ordered operation of the *Start*, *Stop*, *Resume*, and *Terminate* operators, where the sub-sequence including the *Stop* and *Resume* operators may be applied several times:

```
Start( pipeline )
. . .
Stop( pipeline )
. . .
Resume( pipeline )
. . .
Stop( pipeline )
. . .
Resume( pipeline )
. . .
Terminate( pipeline )
```

Clearly, the *Terminate* operator may not be invoked in the previous sequence, if a complete Pattern Instance's execution is required.

The above Behavioural Operator sequence defines the possible *execution states* for a *Pattern Instance*: through the *Start* operator a non-running Pattern Instance changes to the "executing" state; the *Stop* operator causes a transition from the "executing" state to the "suspended" state; the transition from the "suspended" state back to the "running" state is operated by the *Resume* operator; and finally, the *Terminate* operator forces a transition (either from the "running" or "suspended" states) to the "terminated" state, to which the *Start* operator may be applied again. Other possible common sequences are: *Repeat* and *TerminateRepeat*; *Restart* and *TerminateRestart*; *Log* and *TerminateLog*; and *SeqLog* and *TermSeqLog*. In each of these sequences, the time between the two operators' invocations is user defined.

4.5.2 Controlling Individual Executions in the Context of the Restart/Repeat Operators

This section highlights the possible results of combining the *Restart* or the *Repeat* operators with other *Execution Operators*.

I – Usage of the Terminate Operator

The *Terminate* operator terminates a single pattern's instance execution, whereas the *TerminateRestart* and *TerminateRepeat* operators discontinue the action of the *Restart* and *Repeat* operators, respectively. Consequently, if it becomes necessary to abort the current execution of a Pattern Instance being ruled by the two latter operators, the user may call the *Terminate* operator meanwhile. Therefore the following sequences:

```
Restart( time_interval, pipeline )
```

```
. . .
```

```
Terminate( pipeline )
```

```
. . .
```

and

```
Repeat( n, pipeline )
```

```
. . .
```

```
Terminate( pipeline )
```

```
. . .
```

will interrupt the current pattern instance's execution, but will not discontinue the action of the *Repeat* and *Restart* operators.

II – Usage of the Stop/Resume Operators

Similarly to the previous examples, and considering that a single execution of a particular pattern instance lasts enough time to be controlled, the user may also apply the *Stop* and *Resume* operators to each of the several individual executions generated as a result of the *Repeat* and *Restart* operators. This means that the following sequences are also valid:

```
Restart( time_interval, pipeline )
```

```
. . .
```

```
Stop( pipeline )
```

```
. . .
```

```
Resume( pipeline )
```

```
. . .
```

and

```
Repeat( n, pipeline )
```

```
. . .
```

```
Stop( pipeline )
```

```
. . .
```

```
Resume( pipeline )
```

```
. . .
```

III – Usage of the Limit Operator

Since both the *Repeat* and *Restart* operators make use of the *Start* operation, the *Limit* operator may also be applied to define a maximum amount of time for one individual execution of the particular pattern instance being operated. However, and according to the semantics of the involved operators previously defined, different results may be produced. First, the definition of the following *Compound Behavioural Operator*:

```
Limit( time_interval, Repeat( n, pipeline ) )
```

```
. . .
```


might: a) limit the execution time of the first execution of the pattern instance, in case the time interval is less than the time the first execution of the pipeline pattern would take to be completed; or b) interrupt the execution of the second or a subsequent invocation resulting from the *Repeat* operation, in case the time interval is higher than the execution time of the first execution. In either case, the activity of the *Repeat* operator would not be discontinued.

Second, the user may also invoke a Behavioural Operator sequence like:

```
Restart( time_interval, P )
( . . . )
Limit( time_interval, pipeline )
. . .
```

where the particular pattern instance's execution iteration that may be interrupted as consequence of the *Limit* operator would also depend on the total time of each individual execution of the pattern instance, and on the time interval defined by the user as argument to the *Limit* operation.

Finally, it might also be desirable to use the *Limit* operator to control the execution time of the individual execution iterations that are generated by the *Repeat* and *Restart* operators. Such might be possible if instead of calling the *Start* operator to launch each iteration, the *Repeat* and *Restart* operators would call instead a *Compound Behavioural Operator* in the form *Limit(δT , Start(P))*. Such improvement, however, has to be deferred to future versions of our model.

IV – Combining the Restart and Repeat Operators

Sequences of the *Restart* and *Repeat* operators should be used with care, as their semantics were defined separately. Specifically, those operators may be applied in sequence as long as the necessary time interval argument in *Restart* and the overall time to do "n" iterations in *Repeat* do not conflict. Namely, the sequence:

```
Repeat( n, pipeline )
. . .
Restart( time_interval, pipeline )
. . .
```

is safe if the overall time for the "n" iterations terminates before the *Restart* is invoked. Likewise, the sequence

```
Restart( time_interval, pipeline )
. . .
Repeat( n, pipeline )
. . .
```

is also possible if the *Repeat* is invoked and completes in between two consecutive restarts of the pattern instance's execution controlled by *Restart*.

However, to prevent disruptive mixed execution launches provoked by the sequential application of the above two operators, their semantics should have been commonly defined in terms of the current state of the operated pattern instance. A state machine within that pattern

instance's context would define, for example, that a restart iteration would be ignored if a repeated execution was under way, or that a call to the *Repeat* would only imply "n-1" iterations, in case one execution was already occurring as a result of *Restart* operator.

Nonetheless, it might have been desirable to provide a *Compound Behavioural Operator* like *Restart(δT , Repeat(n , P))* that, at each restart of the "P" pattern instance's execution, "P" would be executed "n" number of times according to the *Repeat* operation. The possibility of such composition may be eventually made available in future versions of our model.

Finally, and considering situations where sequential invocations of the *Repeat* operator may occur, such sequence should be avoided if the full execution resulting from invoking the operator the first time does not terminate before the second invocation. The same applies to two sequential invocations of the *Restart* operator.

4.6 Summary

This chapter described the semantics of the Structural and Behavioural Operators for Pattern manipulation. Such description included a few application examples on operator usage.

5

Towards Pattern-based Reconfiguration

Contents

5.1	Introduction	136
5.2	The Methodology Revisited	136
5.3	Reconfiguration	163
5.4	Summary	168

This chapter describes an extension of the methodology towards reconfiguration where patterns are the units for development and run-time reconfiguration.

5.1 Introduction

One important characteristic of our model is the possibility to directly modify patterns in the final application configuration, even at execution time. Such pattern manipulation is possible since patterns remain as first class entities during the whole application development cycle. Therefore, the reconfiguration process is also based on pattern manipulation through operators. Namely, modifications in the application configuration are restricted to the operated patterns, and in the independent dimensions of structure and behaviour. Such implies that the modifications are restricted to the sub-domains represented by each of the operated patterns, even if they are embedded in a Hierarchical Pattern.

Moreover, a pattern-based reconfiguration through pattern operators allows the user to control the reconfiguration process itself. Namely:

- the user may choose which parts of the application are to be modified (i.e. which patterns), and also when to modify them (by using the *Stop/Resume* operators, or the *Terminate/Start* operators);
- the user may also choose to modify only the structure of the application (i.e. by applying the Structural Operators to the selected patterns), or only the behaviour (i.e. by applying the Coordination and Execution Operators); or both (e.g. by replacing one Pattern Instance to another after checking their compatibility through an Inquiry Operator);
- the reconfiguration steps may defined in a operator script (e.g. to be reused, or to be launched automatically);
- pattern operators allow the definition of different reconfiguration policies to modify the same application.

To that extend, this chapter defines how to manipulate a pattern in the different stages of the application development time, including execution time.

In our opinion, the above characteristics may contribute to minimise the overall effects of reconfiguration over a running application, although such still has to be fully validated in our future research. Nevertheless, the main aspects of pattern-based reconfiguration and their potentialities are illustrated in this Chapter. Namely, the first section describes an extension to the methodology discussed in Chapter 3 and defines the foundations for the pattern-based reconfiguration process (concerning development and execution time) which, in turn, is illustrated in the second section.

5.2 The Methodology Revisited

This section presents an extended version of the possible methodology described in section 3.1.3 to represent also different possible ways to build an initial *Application Configuration*, as well as the possibility to modify it at development time. Application modification, in particular, is useful to adapt the configuration to new requirements and, therefore, the considerations discussed in this section are used as the basis to support application reconfiguration as described in section 5.3 ahead.

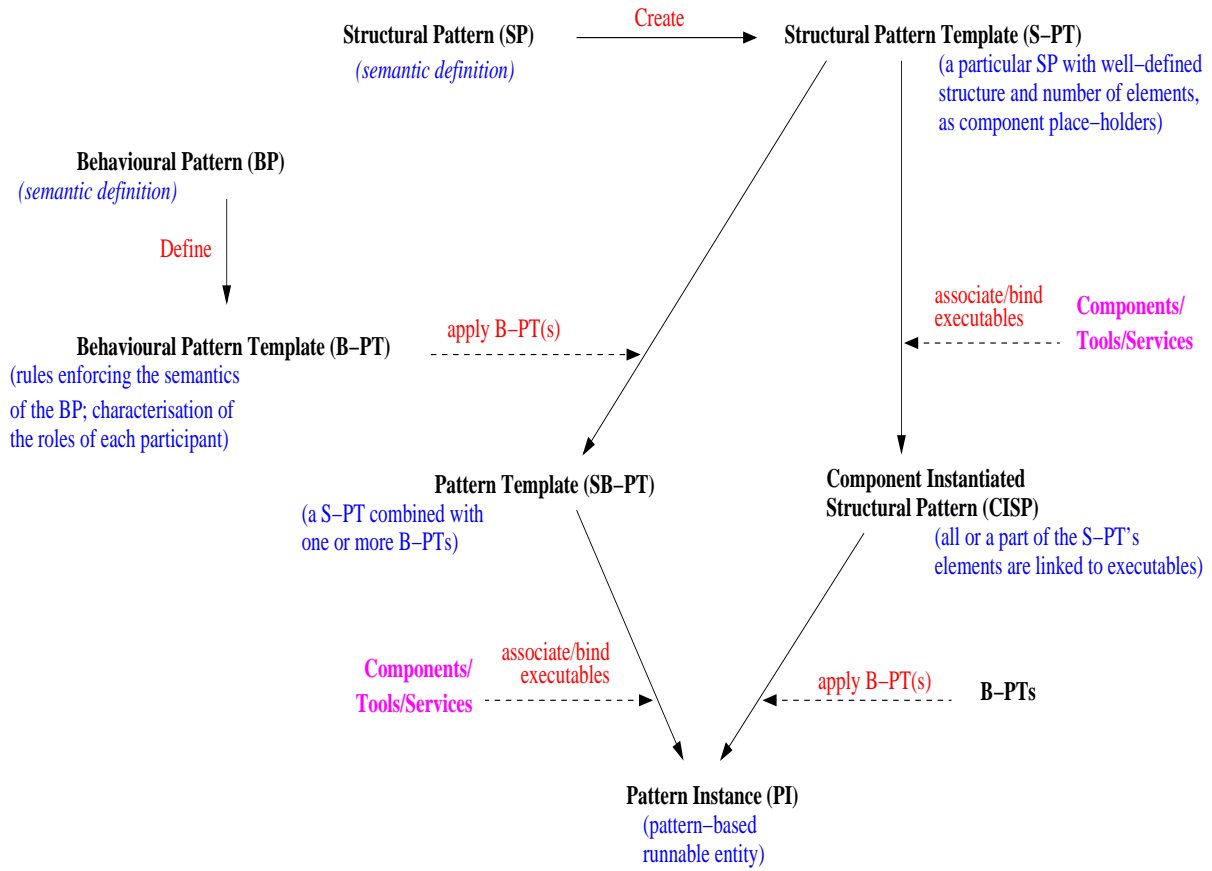


Figure 5.1: Relating the used pattern definitions.

The first sub-section describes the steps of the extended methodology, and the other sub-sections, in turn, discuss operator usage in that context, specifically on handling other possible pattern entities besides *Structural Pattern Templates (S-PTs)* and *Behavioural Pattern Templates (B-PTs)*, as defined in section 3.1.1. For clarity, we re-display the inter-relation between those entities in Figure 5.1. First of all, section 5.2.2 describes the manipulation of *SB-PTs*, i.e. *Structural Pattern Template (S-PT)* combined with one or more *Behavioural Patterns*. Second, the operation of *CISP*, i.e. *Component Instantiated Structural Patterns* is described in section 5.2.3. Finally, section 5.2.4 discusses the handling of *Pattern Instances (PIs)*.

5.2.1 Methodology Steps

Figure 5.2 extends Figure 3.2 with additional ways for application configuration. Next, we describe the set of possible methodology steps represented in the Figure.

First, and similarly to what was previously described in section 3.1.3, the user selects the adequate *Structural Patterns Templates*, and *second*, defines the application's *Structural Configuration* by manipulating those templates through *Structural Operators*.

Third, *Behavioural Pattern Templates* are applied to the elements in the structural configuration, thus producing a *Template Configuration*. If necessary, this configuration is further extended/changed by applying additional *Templates* and *Operators* (both structural and behavioural like *Ownership Operators*).

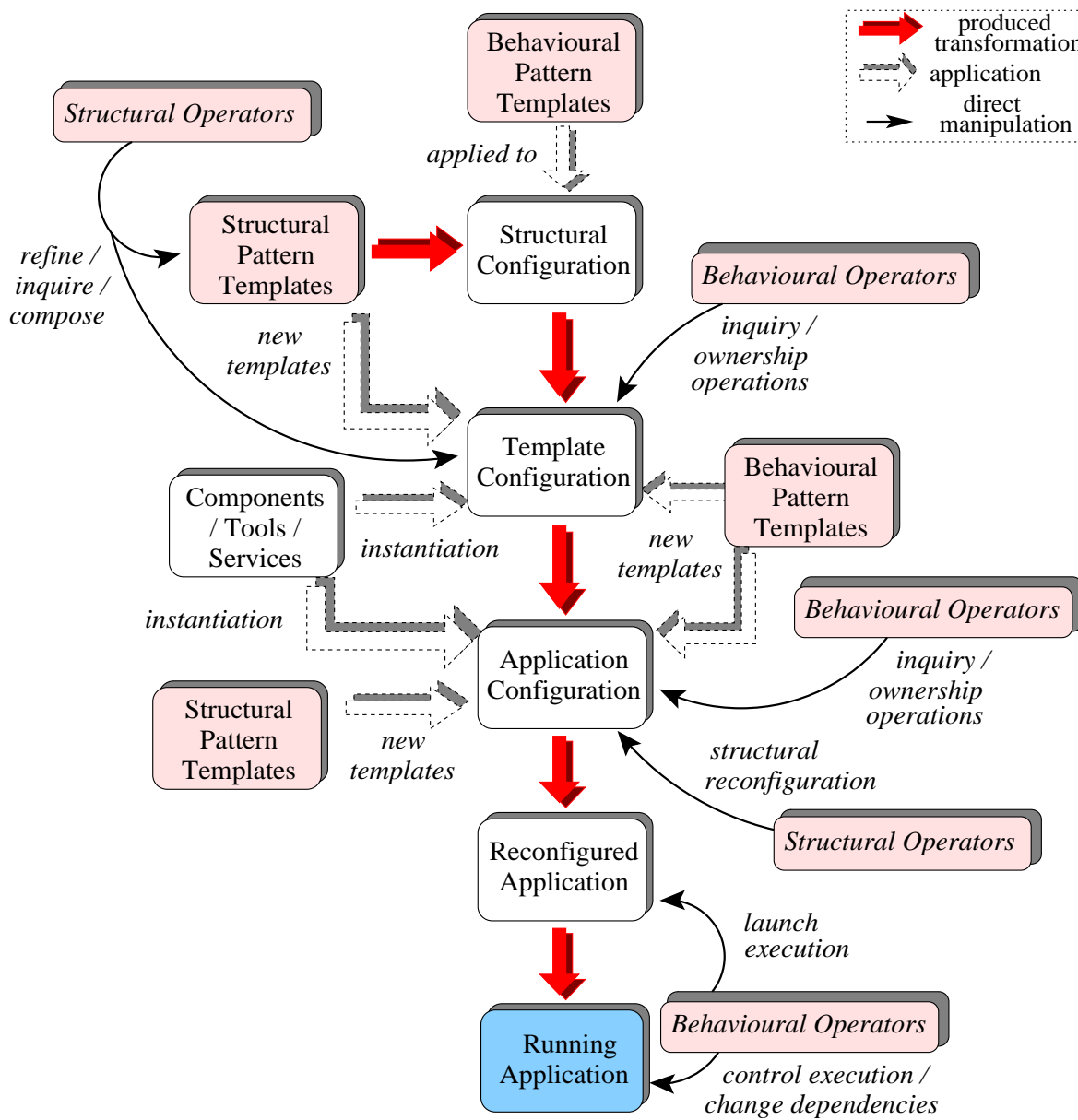


Figure 5.2: Methodology steps for application configuration and execution control.

Fourth, the instantiation of the component place-holders within the *Template Configuration* generates the *Application Configuration*. Unlike the basic methodology presented in section 3.1.3, the user may still change this configuration. Namely, as shown in Figure 5.2, the user may apply other (Structural and Behavioural) Pattern Templates, instantiate the new component place-holders, and activate the necessary operations through the (Structural and Behavioural) Operators. Such changes may be done in the following ways:

- A-** The user recursively applies the previously described sequence: adds and manipulates Structural Patterns through Structural, Inquiry, and Ownership operators; combines new Behavioural patterns to the new *Template Configuration* and operates them through Ownership operators; and finally instantiate the available component place-holders.
- B-** Particular *Pattern Instances*, i.e. fully instantiated (Structural plus Behavioural) *Pattern Templates*, may also be replaced with other Pattern Instances. Before replacement, the user

may check for the compatibility of those Pattern Instances through an *Inquiry Operator*, namely, *IsCompatible(P1, P2)* (section 3.3.3).

- C- The user first associates specific component executables to *Structural Pattern Templates* before defining the (data and control) flow dependencies between them (i.e. before applying particular Behavioural Patterns to those *Structural Pattern Templates*). These patterns' elements are therefore tagged relating them to the associated component executables. However, these resulting *Component Instantiated Structural Patterns (CISPs)* are still not runnable, meaning that they cannot yet be operated by executable operators, although they can be saved in a repository for later reuse. Having defined those *CISPs*, the user may then combine them with the necessary Behavioural Patterns generating *Pattern Instances (PIs)*. Ownership operators may be applied before or after that combination.

Such changes, specifically options “A” and “C” above follow the left and right branches, respectively, displayed in Figure 5.1. In fact, this extended version of the methodology is considered a generalisation of the one presented in section 3.1.3 as a result of the changes “A”, “B”, and “C” which are possible in this fourth step.

As the *final step*, the user launches the application's execution and configures its run-time behaviour through the *Behavioural Operators*, as described in section 3.1.3.

Having identified the new actions concerning a more general possible methodology for pattern-based application configuration, the next subsections clarify some of those steps. The first sub-section (section 5.2.2) describes the third methodology step, namely the operation of a Structural Pattern already combined with one or more Behavioural Patterns. The subsequent sub-sections clarify the semantics of some operators when applied to already instantiated patterns represented in the fourth step above. Specifically, the second sub-section (section 5.2.3) describes examples of operating non-runnable instances of structural patterns, i.e. whose elements have already been linked to some executables of services but for which no flow dependencies have been defined. The third sub-section (section 5.2.4), in turn, describes examples of operating Pattern Instances before executing them.

5.2.2 Operating a Pattern Template (SB-PT)

As mentioned before, the designation *SB-PT* refers to a Structural Pattern Template already combined with one or more Behavioural Pattern Templates.

Moreover, we recall the two ways of combining Behavioural Patterns to a Structural one, as described in section 3.2.5:

- A single Structural Pattern Template is combined with a single Behavioural Pattern Template, i.e all component place-holders are annotated with roles within a unique Behavioural Pattern, and in a well-defined way. We designate this pattern as a *Regular SB-PT*.
- A single Structural Pattern Template is combined with two or more different Behavioural Patterns implying that some component place-holders may be annotated with different roles related to different Behavioural. We define this as an *Heterogeneous SB-PT*.

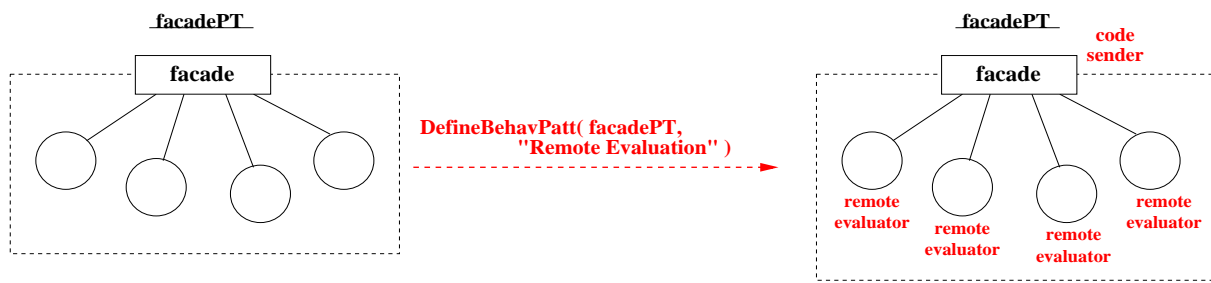


Figure 5.3: Applying a single Behavioural Pattern to all elements of a Structural Pattern forming a Regular SB-PT.

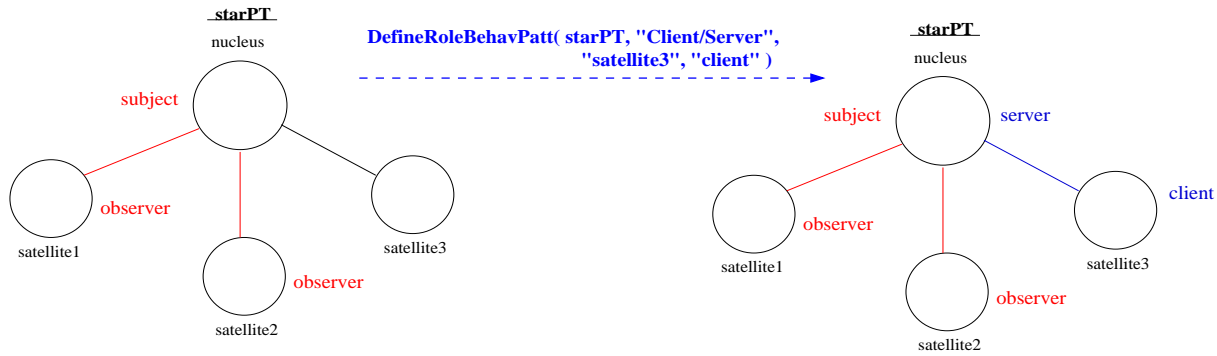


Figure 5.4: Defining the behavioural role of one specific element within a pattern (and the adding of other necessary behavioural annotations). Since that behavioural role pertains a different Behavioural Pattern than the one already applied to the pattern, the result is an Heterogeneous SB-PT.

Two examples were presented in that section:

1. Figure 3.12 showed a *Regular SB-PT* – a Facade S-PT combined with a *Remote Evaluation B-PT*. The generation of this Regular SB-PT is presented in Figure 5.3 as a result of the *DefineBehavPatt(facadePT, “Remote Evaluation”)* operator.
2. Figure 3.13 depicted an *Heterogeneous SB-PT* – a Star S-PT combined with two B-PTs, namely a *Client/Server B-PT* and an *Observer B-PT*. To build such a SB-PT, the user has to annotate each particular element within the the structure with a particular behavioural annotation. Figure 5.4 represents the result of applying the *DefineRoleBehavPatt(starPT, “Client/Server”, “satellite3”, “client”)*, which annotates the element “satellite3” with a different role from the other satellites, thereby generating an Heterogeneous SB-PT.

The operators *DefineBehavPatt* and *DefineRoleBehavPatt* are both *Global Coordination Operators* which were defined in section 3.3.6.

In general, both *Regular* and *Heterogeneous SB-PTs* can be manipulated in three ways:

1. by the application of *Ownership Operators*;
2. through *Structural Operators*;
3. by *changing the behavioural annotations* associated with the Pattern Template’s component place-holders.

Ownership operations are independent from the SB-PTs' behavioural annotations, and therefore are not discussed further. However, for the other two kinds of manipulations, the distinction between *Regular* and *Heterogeneous SB-PTs* allows us to define some optimisations on those patterns' manipulation, namely in what concerns a structural reconfiguration independently from the behavioural annotations, and change of behavioural dependencies independently from the underlying structure.

First of all, we define as possible, the replacement of the Behavioural Pattern Template (B-PT) ruling a *Regular SB-PT* by another B-PT without changing the underlying structure. Moreover, we define that the behavioural annotations of a *Regular SB-PT* may be pre-defined for new component place-holders that may be added to the pattern, according to the applied Behavioural Pattern of the *SB-PT*. This means that structural reconfigurations are possible without changing the existing behaviour (i.e. the behaviour of the older elements is not affected and the behaviour of the new elements is pre-defined). As for *Heterogeneous SB-PTs* no optimisations are assumed, meaning that the user has to explicitly annotate the behaviour of new pattern's elements added to the structure.

To further clarify the above definitions, we start by describing possible *Regular SB-PTs*, and then exemplify structural reconfiguration independent from behavioural definitions, and also how to change these.

I – Regular Pattern Templates (SB-PTs)

Eligible *Regular SB-PTs* are:

- a *Pipeline* or a *Ring* pattern where all elements are coordinated by one of the following Behavioural Patterns: a) the *Peer-to-Peer* pattern; b) the *Streaming* pattern; c) the *Itinerary/-Mobile Agent* pattern where the elements in the chain pre-define the places the agent has to travel to; d) the *Producer/Consumer* pattern; e) the *Client/Server* pattern; f) the *Remote evaluation* pattern where the structural elements define the (path of) host executors for a mobile code.
- a *Star* pattern whose elements may be coordinated by:
 - a) the *Client/Server* pattern where the nucleus is ruled by the "server" role within the pattern, and all present and future satellites are "clients";
 - b) the *Master/Slave* pattern, where the nucleus is the "server" and all the clients obey the "slave" behaviour within the pattern;
 - c) the *Producer/Consumer* pattern with the nucleus being the "producer" and all satellites being "consumers";
 - d) the *Observer* pattern with the nucleus being the "subject" and the satellites the "observers";
 - e) the *Parameter-Sweep* pattern where the nucleus dispatches identical code to all satellites but with different parameter values and then collects the results;
 - f) the *Code-on-demand* pattern where the nucleus requests the code from its satellites;

- g) the *Remote evaluation* where the nucleus sends the requested code to be executed to the satellites;
 - h) the *Streaming* pattern where data flows either from all the satellites to the nucleus or, inversely, from the nucleus to all satellites.
- a *Proxy* pattern whose elements may be coordinated by:
 - a) the *Client/Server* pattern with the RealSubject behaving as the "server" and the proxies as "clients";
 - b) the *Producer/Consumer* pattern where the RealSubject sends data to be consumed by the "proxies" at the desired rate;
 - c) the *Mobile Agent/Itinerary* where the sequence a proxies defines the traveling path of the "agent" (i.e. the *Real Subject*);
 - d) the *Observer* pattern where the proxies are notified of events related to the subjects they are registered to at the *Real Subject*.
 - a *Facade* pattern whose elements may be coordinated by:
 - a) the *Client/Server* pattern where the facade element ("client") forwards requests to the sub-systems (the "servers");
 - b) the *Remote Evaluation* pattern where the facade element sends the code to be executed by one (or more) of the sub-systems,
 - c) the *Code-on-Demand* pattern where the facade element requests the code to be executed from one of its sub-systems;
 - d) the *Observer* pattern where the sub-systems have to observe certain events at the facade element;
 - e) the *Streaming* and *Producer/Consumer* patterns where the subsystems consume the data sent by the facade element,
 - f) the *Master-Slave* pattern where the sub-system elements behave as the "slaves" and the facade element plays the "master" role within the pattern.
 - an *Adapter* pattern whose elements are ruled by: a) the *Client/server* pattern, where the "adaptee" is the "server", and the "adapter" is the client; b) the *Streaming* pattern, where data flows from the "adapter" to the "adaptee", and back from the "adaptee" to the "adapter".

For all the above examples, it is possible to define cases where new added elements will behave in a pre-defined way. For example: a) on adding a new element to a Star S-PT combined with a *Master/Slave* B-PT where the "nucleus" is annotated with the "master" role, the new satellite is to be automatically annotated with the "slave" role within that Behavioural Pattern; b) on applying the *Extend* operator to an *Adapter* S-PT ruled by a *Client/Server* B-PT (where the original "adaptee" element is the "server", and the "adapter" is the "client"), the new "adapter" is to be annotated with the "client" role, whereas the old "adapter" becomes its "server", while remaining the "client" of the original "adaptee".

Moreover, it would also be possible to define a map table between pairs of some of the above *Regular SB-PTs* establishing which *Regular SB-PTs* can be transformed into other *Regular SB-PTs*, in the dimensions of behavioural modification or structural transformation. For instance, examples of both conversions are, respectively: a Ring S-PT combined with the *Peer-to-Peer* B-PT can be transformed into the same Ring S-PT combined with the *Streaming* B-PT; the original Ring-PT combined with the *Peer-to-Peer* B-PT can also be transformed into a Pipeline S-PT (with the same number of elements) combined with the same *Peer-to-Peer* B-PT. Mappings like these support optimisations on both structural and behavioural transformations, some of which will be explained in examples in the sub-sections ahead, and also in the manipulation of both *CISPs* and *PIs*.

To conclude, and on the definition of *Regular SB-PTs* through the application of the *DefineBehavPatt*(*P*, *B-P*) to a S-PT, a final remark is due. For some of the *Regular SB-PTs* described above, it is possible to define different ways of applying the same B-PT to a S-PT, and the result is still a *Regular SB-PT*. For example, on applying the *Streaming* Behavioural Pattern to the *Star* Structural Pattern, two different combinations are possible, both resulting in *Regular SB-PTs*. Specifically, in case data flows from the nucleus to all satellites, if another satellite is added, it will also receive data from the nucleus; in case the nucleus receives data from all satellites, it will also receive data from any other satellite added to the *Star* SB-PT. However, the *DefineBehavPatt*(*StarS-PT*, “*Streaming*”) operator does not provide a way to distinguish between those cases.

As defined in section 3.3.6, the *DefineBehavPatt*(*P*, *B-P*) assumes that the mappings between the elements of “*P*” and the “roles” within “*B-P*” are pre-defined and are implementation dependent. Therefore, it is up to the implementation to somehow distinguish between the different possibilities of applying the same Behavioural Pattern to a S-PT, to form a *Regular SB-PT*. However, in order to make such distinction, an extra parameter could also have been added to the definition of the *DefineBehavPatt* operator (and also to the *ReplaceBehavPatt* operator) that would represent information associated to the specific “*B-P*”. This will be included in a future version of our model.

For the time being, and for simplification reasons, whenever the application of the *DefineBehavPatt* operator may generate any ambiguities, these will be explicitly clarified in the text. Nevertheless, most *Regular SB-PTs* defined above clearly specify which is to be the behavioural role of each element in the operated S-PT.

Next section describes possible structural operations on SB-PTs, either *Regular* or *Heterogeneous* SB-PTs.

II – Structural Operation of SB-PTs

In this section we define two possible structural manipulations of both *Regular* and *Heterogeneous* SB-PTs:

1. Structural Operators act only over the structure of a SB-PT with no regard to the behavioural annotations.
2. A SB-PT may be structurally manipulated as whole (i.e. a first class entity) and, consequently, the behavioural annotations are also taken into account. This allows further optimi-

sations for some Structural Operators. To distinguish such manipulation from the previous one, we use the designation *FCSB-PTs* (i.e. *First Class (Structural plus Behavioural) Pattern Templates*) for a SB-PT whenever necessary.

Both types of structural manipulations will be exemplified for each of the following operators where the distinction between *Regular* and *Heterogeneous* SB-PTs will be highlighted whenever relevant.

1 – Replicate operator

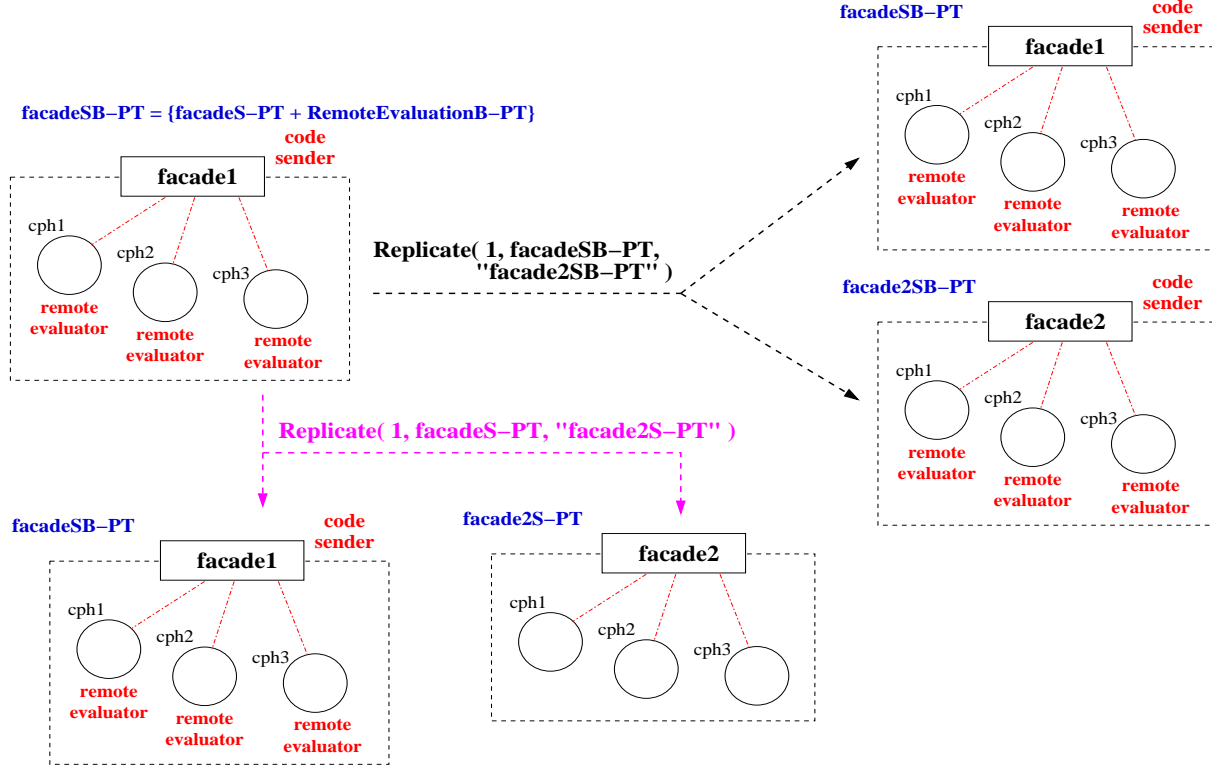


Figure 5.5: Replicating a SB-PT in two ways: a) considering it as a first class entity ("facadeSB-PT"); b) acting only over the Structural Pattern Template included in the SB-PT ("facadeS-PT").

The duplication of SB-PTs, either *Regular* or *Heterogeneous*, by the *Replicate* operator can be done in two ways:

- if a SB-PT is a FCSB-PT, its defined structure and applied behavioural annotations are duplicated for the replicas, meaning that identical FCSB-PTs are created. This is exemplified on Figure 5.5 where the FCSB-PT named "facadeSB-PT" is replicated once through *Replicate(1, facadeSB-PT, "facade2SB-PT")* generating a new SB-PT named "facade2SB-PT". Otherwise,
- only the structure of the SB-PT is duplicated creating a new Structural Pattern Template. This is done on bottom of Figure 5.5 by applying the *Replicate* operator to the S-PT within the pattern "facadeSB-PT". As represented in the Figure, this SB-PT is the combination of a Facade Structural Pattern Template named "facadeS-PT" and a Remote Evaluation Behavioural Pattern Template named "RemoteEvaluationB-PT". The operation *Replicate(1, facadeS-PT, "facade2S-PT")* explicitly acts upon the structure of the SB-PT, and produces a new S-PT named "facade2S-PT".

In this example, as well as from now on, we use different identifiers for a SB-PT and its associated S-PT and B-PT, as a way to distinguish between the access to the SB-PT as whole (i.e. treating it as a first class entity, i.e. FCSB-PT), or to the S-PT/B-PT, individually. Nonetheless, such distinction could have been exemplified by adding an extra parameter to the definition of the operators (e.g. a flag) that would differentiate if the SB-PT is to be structurally manipulated as a FCSB-PT or not.

2 – Replace and Reshape operators

On one hand, the *Replace* operator can be applied to a FCSB-PT replacing it by another FCSB-PT. For example, it is possible to replace a FCSB-PT like a *Proxy* Structural Pattern Template (S-PT) annotated with a *Master-Slave* Behavioural Pattern Template (B-PT) which is included in a *Hierarchical Pattern* by another FCSB-PT like a *Star* S-PT annotated with the *Client/Server* B-PT.

On the other hand, the *Replace* operator may also be used to replace the Structural Pattern Template (S-PT) within a SB-PT by another S-PT while keeping the Behavioural Pattern Template (B-PT), resulting on a new SB-PT. For example, it could be possible to replace the *Star* S-PT in the previous example (i.e. a SB-PT including a *Star* S-PT and a *Client/Server* B-PT), by a *Proxy* S-PT. However, this feature is only applicable in the context of the *Regular SB-PTs* as the ones described in the previous section, since the role of each element within the SB-PT is regular/pre-defined (e.g. all created proxies become annotated with the client role in the example above, and the Real Subject becomes the server). However, whereas for *Regular SB-PTs* it would be possible to build a mapping (to be automatically applied) of possible structural replacement within a SB-PT, for *Heterogeneous SB-PT* such mapping would be infeasible.

In general, and due to its semantics restrictions previously described, we define the *Reshape* operator not to manipulate SB-PTs.

3 – Embed/Extract operators

To include a SB-PT into another (or into a simple S-PT) through the *Embed* operator, that SB-PT is manipulated as a FCSB-PT. Therefore, the pattern to be embedded keeps the same behavioural annotations as before. The same applies for the *Extract* operator.

4 – Group/Ungroup and Eliminate operators

The *Group* and *Ungroup* operators handle SB-PTs as FCSB-PTs: the grouped FCSB-PTs are represented by the resulting aggregate as a whole, as for normal S-PTs. The *Ungroup* dissolves the FCSB-PT passed as argument but the inner SB-PTs continue to exist, similarly to what was described for S-PTs in section 4.2.1.

The *Eliminate* operator, in turn, when applied to any kind of SB-PTs deletes the Structural Pattern and, consequently, the behavioural annotations.

5 – Increase and Decrease operators

The *Increase* operator can either: a) act over the Structural PT of a SB-PT; or b) manipulate the SB-PT as a FCSB-PT. In the first case, the structure of the SB-PT is increased and the new component place-holders do not have any automatically associated behaviour. Consequently, dependencies for the new place-holders have to be explicitly annotated by the user.

In the second case, since the SB-PT is considered a *first class entity* (FCSB-PT), the control and data flow dependencies of the new component place-holders are pre-defined, as long as the SB-PT is a *Regular FCSB-PT* as described above. In case of an *Heterogeneous FCSB-PT*, the user also has to explicitly annotate the dependencies for the new elements.

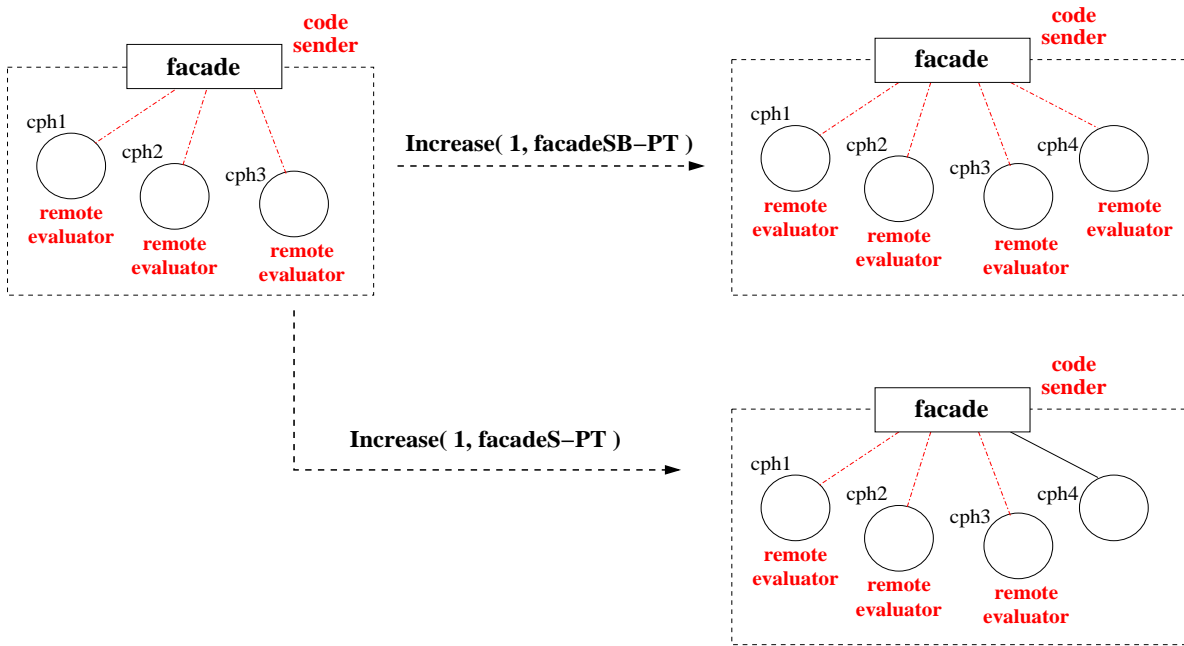


Figure 5.6: Augmenting the number of component-place holders of a SB-PT in two ways: a) considering it as a first class entity ("facadeSB-PT"); b) acting only over the Structural Pattern Template included in the SB-PT ("facadeS-PT").

Figure 5.6 presents the increasing of a *Regular FCSB-PT* concerning the above two cases. As shown in the left side of the Figure, the FCSB-PT is named "facadeSB-PT", and consists of a *Facade S-PT* (identified as "facadeS-PT") combined with a *Remote evaluation B-PT*. The "facade" component place-holder within the FCSB-PT is annotated with the "code sender" role within the B-PT, and the sub-systems are annotated as "remote evaluators".

The top right-hand side of Figure 5.6 shows the result of operating the "facadeSB-PT" as a first class entity – one extra element is added ("cph4") and it is automatically annotated with the "remote evaluator" behaviour. The bottom right-hand side, in turn, shows the direct modification of the structure (identified as "facadeS-PT") ignoring the associated B-PT, resulting on the new component place-holder ("cph4") having no behavioural annotations.

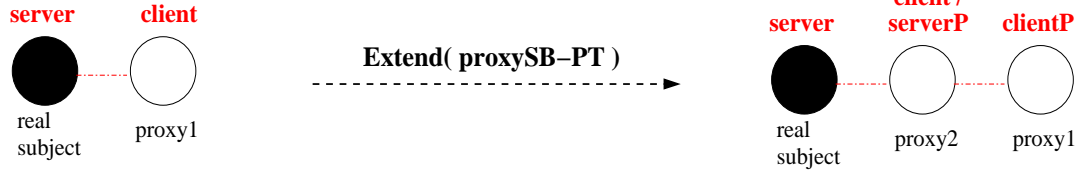
The *Decrease Structural Operator*, in turn, when applied to a SB-PT, either considered as first-class entity or not, removes component place-holders, and eliminates the related behavioural annotations. This has no major implications on the behavioural annotations of the other component place-holders for *Regular SB-PT*. For example, removing a sub-system of the "facadeSB-PT" in the previous example (Figure 5.6) has no effect on the other sub-systems.

However, for the elimination of elements from an *Heterogeneous SB-PT*, it is necessary to explicitly identify the element to be removed. To accomplish such removal, one needs to use the second version of the definition of the *Decrease* operator as presented in section 3.3.2. Moreover, the behavioural annotations of the remaining component place-holders after the *Decrease* operation may have to be explicitly modified by the user (e.g. the removal of a pipeline's inner stage that behaves as a server to the previous client stage and as a producer to the subsequent consumer stage leads to an incoherent result).

A SB-PT whose underlying structure obeys the *Adapter* Structural Pattern requires an additional remark – the restriction concerning the impossibility of operating an *Adapter* S-PT remains. Therefore, the *Increase/Decrease* operators are not applicable to such a SB-PT.

6 – Extend and Reduce operators

$\text{proxySB-PT} = \{\text{proxyS-PT} + \text{Client/ServerB-PT}\}$



$\text{facadeSB-PT} = \{\text{facadeS-PT} + \text{RemoteEvaluationB-PT}\}$

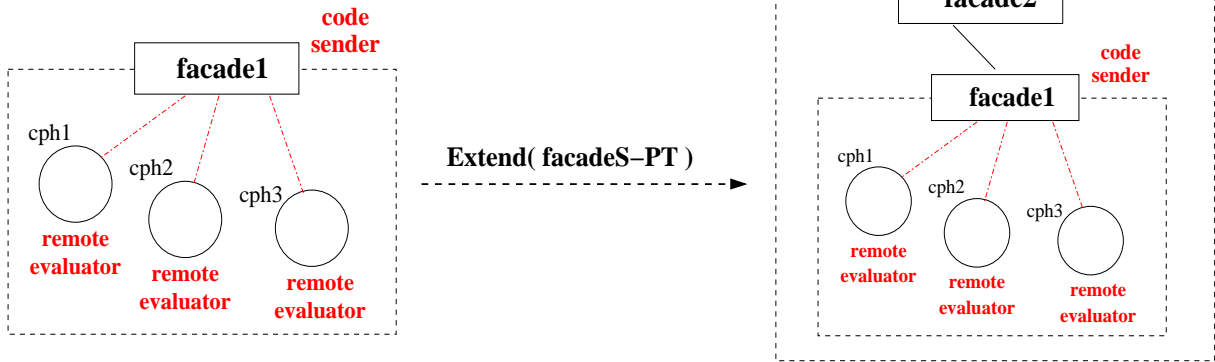


Figure 5.7: Extending two SB-PT in two ways: a) considering it as a first class entity (“proxySB-PT”); b) acting only over the Structural Pattern Template included in the SB-PT (“facadeS-PT”).

Similarly to the *Increase* operator, the *Extend* operator can also either: a) act over the Structural PT of a SB-PT; or b) manipulate the SB-PT as a FCSB-PT. Figure 5.7 presents two examples concerning those situations. As an instance of the first situation (a), and on bottom of the Figure, the underlying structure of the pattern “facadeSB-PT” (a *Facade* S-PT combined with a *Remote evaluation* B-PT) is augmented through the *Extend* Structural operator. The access to that structure is done through “facadeS-PT”, passed as argument to *Extend* and, consequently, a new component place-holder is added to the structure (according to what was defined for the *Facade* Structural Pattern (Figure 4.6 in section 4.2.1)), and no behavioural annotations are added to the new element “facade2”.

As an example of the second situation (b), the top of Figure 5.7 shows the *Extend* operator being applied to a *Proxy* S-PT whose elements are annotated with roles within the *Client/Server* Behavioural Pattern. The “proxySB-PT” in the top left-hand side of the Figure includes a “proxy1” component place-holder, which is annotated as the “client” within the *Client/Server* pattern, and the “realsubject” which is associated with the “server” role. Since this SB-PT is operated as a first class entity, structural modifications are automatically annotated with data and control flow dependencies. Consequently, extending the “proxySB-PT” results in the addition of an extra component place-holder – “proxy2” – that is tagged with two roles. On one side, “proxy2” becomes the (direct) “client” of “realsubject”. On the other side, “proxy2” is also

annotated as a server (identified as “serverP”) to the original “proxy1” (symbolically annotated as “clientP”). In this way, “proxy2” can forward normal requests from “proxy1” to be answered by the “realsubject”. The present example may illustrate the migration of a service to a new location resulting in a chain of proxies to reach that service.

proxySB-PT = {proxyS-PT + Client/ServerB-PT}

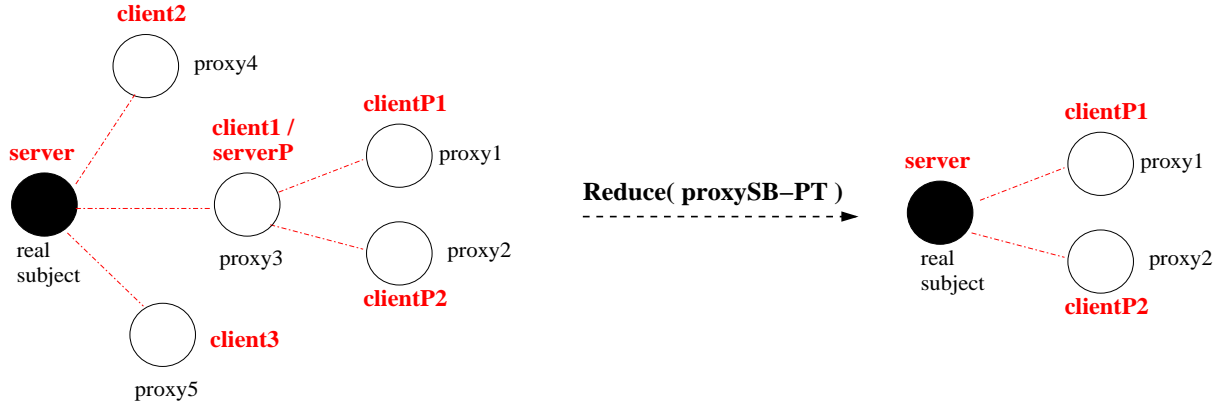


Figure 5.8: Applying the Reduce operator to a SB-PT.

As for the application of the *Reduce* operator to a SB-PT, it is not necessary to distinguish between operating only the structure of a SB-PT and operating it as *FCSB-PT*. The *Reduce* operation results in the elimination of some elements according to the semantics of the operated S-PT (e.g. Figures 4.29 and 4.30 in section 4.3), and consequently, all related behavioural annotations are also eliminated.

For example, Figure 5.8 illustrates the application of the *Reduce* to a Proxy S-PT combined with the *Client/Server* B-PT (“proxySB-PT”). The “proxySB-PT”, on the left-hand side of the Figure, had already been operated by the *Extend* and *Increase* operators which resulted on three new elements annotated with roles within the applied Behavioural Pattern: “proxy3” represents the “realsubject” acting as a “server” to the pre-existent “proxy1” and “proxy2” elements; “proxy3” is also a “client” to the “realsubject” analogously to the “proxy4” and “proxy5” elements which were created as a result of the *Increase* operator. The operation of *Reduce* on “proxySB-PT” undoes the above actions of the sequential operation of the *Extend* and *Increase* operators, and the necessary behavioural annotations are removed. The result is a contracted structure where the “realsubject” returns to its original position within the structure, thereby replacing “proxy3” which is eliminated, and the elements “proxy4” and “proxy5” are also deleted.

To finalise the discussion on SB-PT manipulation, next section describes how to change the behavioural annotations within a Template Configuration independently from its structure.

III – Behavioural Modification of SB-PTs

A behavioural modification of a Structural PT combined with one or more Behavioural PTs (i.e. a SB-PT) comprises the change of the behavioural annotations of the SB-PT’s component placeholders. Similarly to the structural reconfiguration of *FCSB-PTs* (i.e. SB-PTs manipulated as first-class entities), the modification of the behavioural annotations within those pattern tem-

plates can also be optimised. Specifically, we define the possibility of replacing the Behavioural PT (B-PT) associated to a FCSB-PT by another B-PT, resulting on the automatic modification of all behavioural annotations of the entire component place-holders within the SB-PT.

Nevertheless, we highlight that such automatic behavioural modification is defined in the context of *Regular SB-PTs*, since it is possible to define structural and behavioural transformation mappings between those FCSB-PTs towards reconfiguration, as it was previously discussed in section 5.2.2.

The replacement of a Behavioural PT within a FCSB-PT is defined by the *ReplaceBehavPatt*(*SB-PT*, *B-P1*, *B-P2*) operator introduced in section 3.3.6. The first argument to the operator is to be the first-class SB-PT whose behavioural annotations are to be modified. These are represented as a whole by the second argument, “B-P1”, that identifies the Behavioural Pattern Template to be replaced with the new Behavioural Pattern Template passed as the third argument, namely “B-P2”.

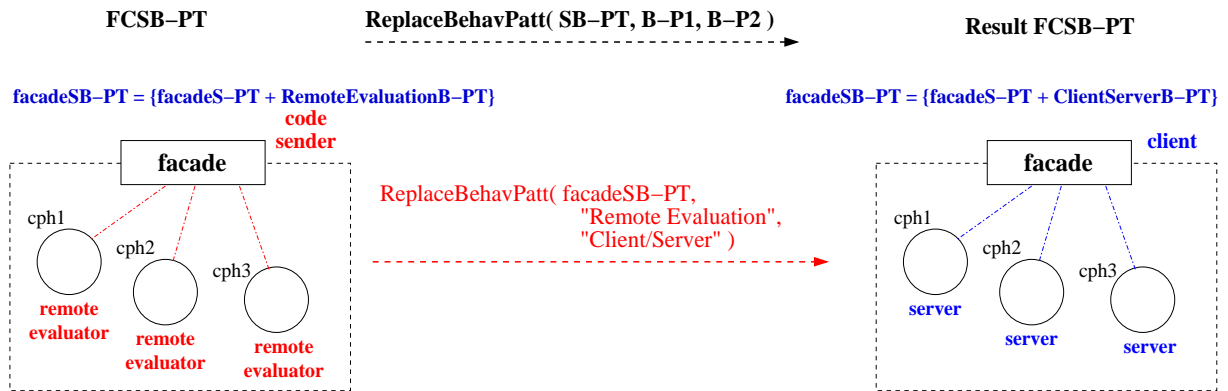


Figure 5.9: Modifying the behavioural annotations of a SB-PT considered as a first class entity (“facadeSB-PT”).

Figure 5.9 shows an example of applying the *ReplaceBehavPatt* operator to the “facadeSB-PT” regarded as a first class SB-PT. This FCSB-PT was previously presented in Figures 5.6 and 5.7. The original behavioural annotations within that FCSB-PT are conform to the *Remote Evaluation* Behavioural Pattern, and the new annotations comply to the *Client/Server* Behavioural Pattern as described in section 5.2.2 – the “facade” component place-holder is tagged with the “client” role and all sub-systems are annotated with the “server” role within that behavioural pattern.

Having described the third step of the more generic methodology explained in section 5.2.1, the following sub-sections (5.2.3 and 5.2.4) describe the fourth step within that methodology. The next-subsection, in particular, considers the manipulation of Structural Pattern Templates still not associated to any Behavioural Pattern Templates, but whose component place-holders are already instantiated to specific executables (i.e. *CISPs*), whereas the subsequent sub-section presents *PI* manipulation.

5.2.3 Operating Component Instantiated Structural Patterns (CISPs)

We define as *Component Instantiated Structural Pattern (CISP)*, a *S-PT* where *all* or a *sub-set* of its component place-holders are already associated to executables. We designate the first case as a *Full CISP*, and the second case as a *Partial CISP*.

To represent the instantiation process we define two new (Structural) Operators, namely:

Instantiate(P, position, component) The component place-holder (CPH) of pattern “P” identified by the parameter “position” is instantiated to the executable/service identified by the parameter “component”. As a consequence, the pattern’s bound element is annotated with the executable/service to be executed at run-time, and can thereafter be accessed by the name of that executable/service (i.e. the value of the parameter “component”).

Unstantiate(P, position) The instantiated element of “P” identified by the parameter “position” is unbound, meaning that the annotations that associate that element to an executable/service are deleted. Whether this un instantiation operation generates a free CPH with the same identifier it received when it was first created, or receives a new name is implementation dependent. Nevertheless, the new name as well as all members’ identifiers should be kept unique within the pattern.

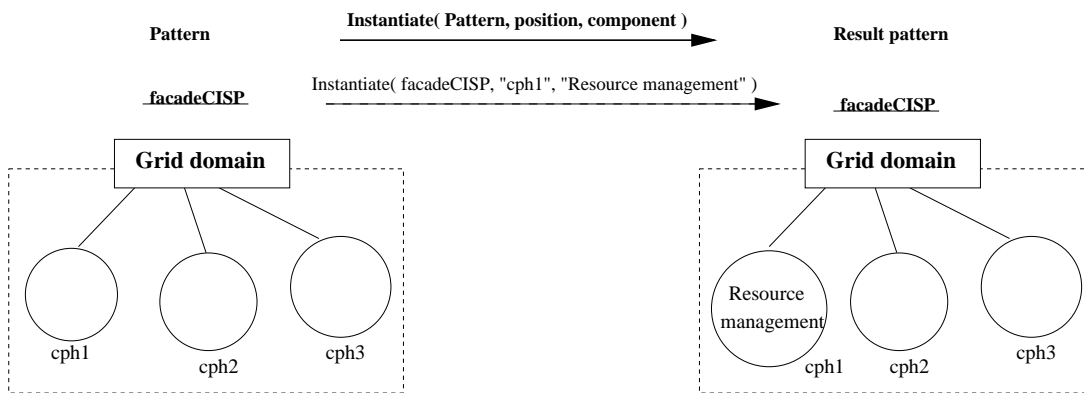


Figure 5.10: Instantiation of the component place-holder “cph1” of the pattern “facadeCISP” to the “Resource management” component.

Figure 5.10 represents, as an example, the association of one of the elements of the pattern “facadeCISP”, namely the component place-holder named “cph1”, to the “Resource management” executable/service. The *Unstantiate(facadeCISP, “Resource management”)* operator would, in turn, eliminate that association resulting on a free component place-holder.

Considering now the semantics of the Structural Operators characterised in Chapter 4, their application to both types of *Component Instantiated Structural Patterns* (i.e *Full CISPs* and *Partial CISPs*) implies no further considerations for the majority of operators. Namely, the *Eliminate*, *Replicate*, *Replace*, *Extend*, *Reduce*, *Embed*, *Extract*, *Group*, and *Ungroup* operators are applied in the same way to CISPs as they were before to *S-PTs*. However, the *Embed* operator can only be applied to CISPs with free (not instantiated) component place-holders.

As an example of the use of one of those operators, Figure 5.11 shows the result of applying the *Extend* operator to an already instantiated Facade Structural pattern. The Facade interface

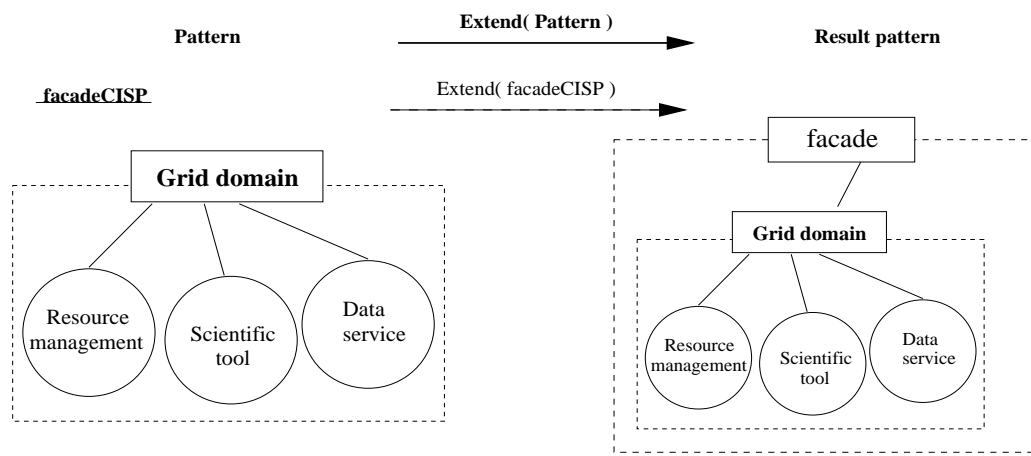


Figure 5.11: Extending a Component Instantiated Structural Pattern, namely "facadeCISP"

is named "facadeCISP" and the pattern represents the controlled access to a particular Grid domain and its available facilities, namely, for resource management, control of a scientific tool, or data services like file transfer. In case this Grid domain is to be part of a new wider Grid domain that will be responsible to control the access to the former domain as well as other Grid domains, the original Facade is manipulated in the usual way through the *Extend* operator. The result of this operation is depicted on the right side of Figure 5.11. The resulting extended Facade can then be increased as necessary, new patterns may also be embedded, and the available component place-holders can therefore be instantiated.

As for the remaining operators, namely *Reshape*, *Increase*, and *Decrease*, further clarifications are needed. The *Reshape* operator, in particular, is not to be applied to CISP as this would imply several particular cases. Nevertheless, this limitation may be revised in further developments of our model. The *Increase/Decrease* operators, in turn, are not applicable to a CISP conforming to the *Adapter* Structural Pattern, similarly to what was defined for an *Adapter* S-PT, but can be applied to the remaining CISP. Such is described in the next sub-sections. Nevertheless, we recall that the *Increase/Decrease* operators are not recursive, meaning that their effect is restricted to the first level of a CISP.

Applying the Increase Operator to a CISP

As it was presented in section 3.3.2, there are two possible versions for the *Increase* operator, namely:

Increase(n, P) Adds "n" new component place-holders to pattern "P" according to its semantics.

Increase(n, P, position) Adds "n" new component place-holders to pattern "P" but at a specific position within the pattern. Such position is related to the pattern element identified by parameter "position".

The version *Increase(n, P)* was previously described for S-PTs in section 4.2.1, and can be applied to the basic cases of the *Star*, *Proxy*, and *Facade* CISP with similar results as if they were S-PTs, due to those patterns' structural characteristics.

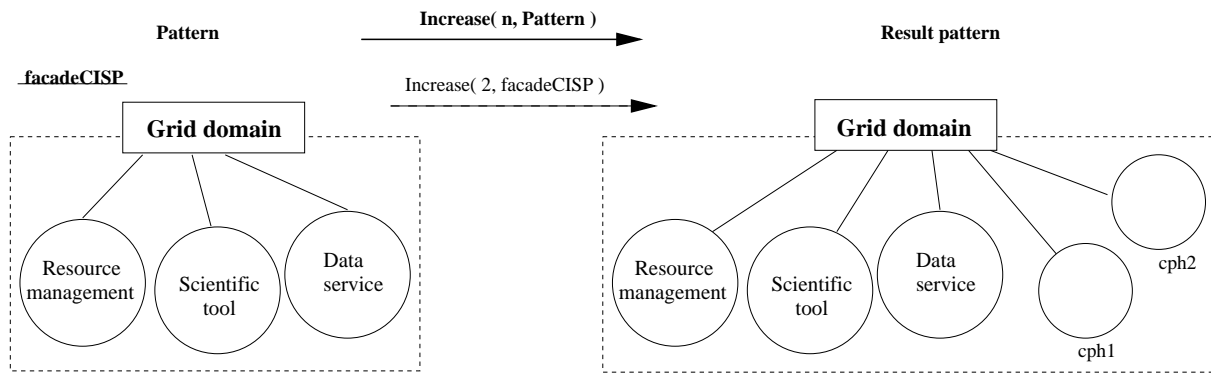


Figure 5.12: Increasing a Component Instantiated Structural Pattern (“facadeCISP”) by two component place-holders.

For example, Figure 5.12 shows the application of the $\text{Increase}(n, P)$ version to a *Full CISP* based on the *Facade* Structural Pattern, i.e. “facadeCISP” which represents a Grid domain providing access to a set of services. The result of the *Increase* operation is the creation of two new component place-holders within the pattern’s structure, and the new elements are labeled with unique tags within that pattern, namely “cph1” and “cph2”. The creation of two new component place-holders for *Star CISP*s or *Proxy CISP*s would be similar as presented for the *Facade CISP*.

However, for cases of the *Pipeline* and *Ring CISP*s, for which it is possible to identify a sequential order for the elements, an extra parameter for the operator is mandatory. Specifically, to identify where to create the new component place-holders within the sequence of elements that define those CISP.s. Therefore, the second form of the *Increase* operator has to be used in those cases. Concretely, the “position” parameter in the $\text{Increase}(n, P, \text{position})$ operator identifies the element within the *Pipeline* or *Ring CISP*s *after which* the new component place-holders are to be placed. In the particular case of a *Pipeline CISP*, if the “position” parameter is instantiated with a “zero” identifier, the new component place-holders are created *before* the *first element* in the pipeline.

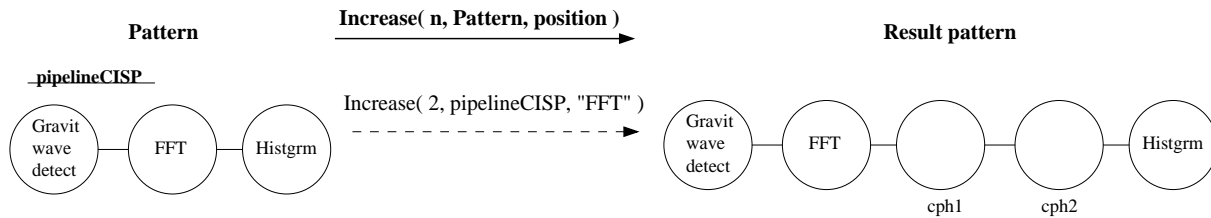


Figure 5.13: Increasing a component instantiated Pipeline (“pipelineCISP”) by two component place-holders inserted after element “FFT”.

For instance, Figure 5.13 shows a case of increasing a *Pipeline CISP* with the definition where to place the new elements. The pattern, named “pipelineCISP”, represents a simple example in the area of astrophysics supporting data analysis and processing of out of space waves. First, these are detected by a “Gravitational wave detector” (first stage in the pipeline), they are then modified by “FFT”, a Fast Fourier transformation, and finally the result is analysed at the last

stage through an "Histogrammer" (a graphical displaying unit for rendering input signals). The right-hand side of Figure 5.13 shows the result of the operation of the *Increase* where the position to include the two new component place-holders is after the "FFT" component. Each new component place-holder is tagged with a unique identifier within the pattern, as usual. On the other hand, the addition of two new component place-holders before the first element of the "pipelineCISP", i.e. "Gravitwavedetect", would for example, be accomplished with the invocation: *Increase(2, pipelineCISP, "zero")*.

A final remark concerning the increasing of instantiated Structural Patterns, in particular of *Facade* CISP like the one in Figure 5.12, is due. According to the semantics of the *Facade* Structural Pattern, the member representing the *facade* element within the structure provides an interface to all *sub-systems* members. Therefore, it is assumed that the addition of new CPHs to an already instantiated *Facade* CISP, i.e. the *facade* element is already bound to an executable/service, is only possible if that *facade* element is also possible to provide an interface for the executables/services that are to instantiate the new CPHs. Therefore, the new CPHs added to the "facadeCISP" in Figure 5.12 are supposed to be instantiated with tools/services compatible with the "Grid domain" element.

Applying the Decrease Operator to a CISP

Similarly to the *Increase* operator, the *Decrease* operator also exists in two versions and can be applied both to partially and fully instantiated Structural Patterns (i.e. *Partial* CISPs or *Full* CISPs):

Decrease(n, P) Eliminates free (non instantiated) component place-holders (CPHs) within pattern "P". The number of CPHs to delete is defined by the parameter "n". In case "n" is greater than the number of free CPHs, only those free CPHs are deleted.

Decrease(n, P, position) Eliminates both free and instantiated CPHs from pattern "P". The parameter "n" defines the number of elements to remove, and the parameter "position" defines the element where the deletion starts. In case of patterns for which it is possible to define an ordering within the pattern, e.g. *Pipeline* and *Ring* CISP, the parameter "n" defines the number of elements to delete after, and including, the element defined by the parameter "position". For the other types of patterns, e.g. *Facade*, *Proxy*, and *Star* CISP without an implementation-defined ordering, this version of the *Decrease* operator only removes a single element. Therefore, the parameter "position" identifies the element to remove, and the parameter "n" is to be instantiated with the value one.

Figure 5.14 depicts the usage of the *Decrease* operator in two forms, namely, non-instantiated elements' removal, and elimination of a specific element. The first form is exemplified by operating a *Facade Partial CISP* named "facadeCISP". The operator invocation results in trying to eliminate two component place-holders from the *Facade*, but since only one exists within the pattern, solely this one ("cph1") is in fact removed.

The second form of the *Decrease* operator is exemplified at the bottom of Figure 5.14 – three elements of a partially instantiated pipeline pattern ("pipelineCISP") are deleted from it, starting from (and including) the "Gaussian" element. In case only the "Gaussian" element is to

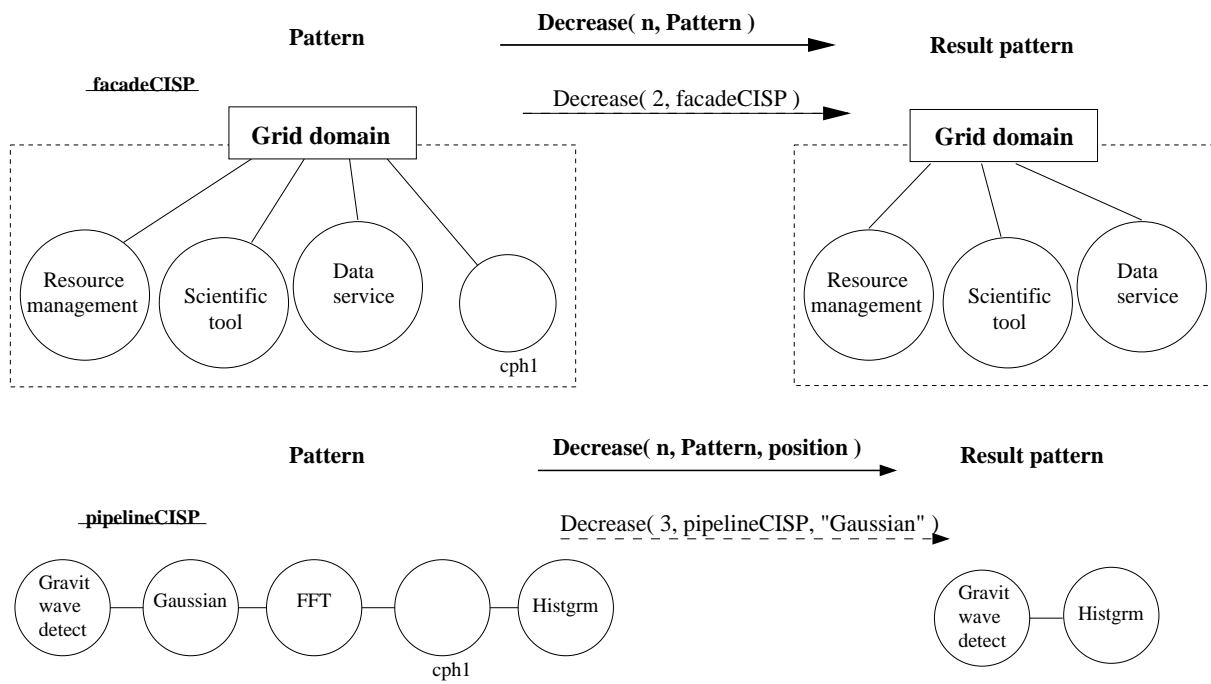


Figure 5.14: Application of the *Decrease* operator. The first example (upper part of the Figure) presents a case of reducing the number of component place-holders from a Partial CISP, namely, a partially instantiated Facade. The second example shows the usage of the *Decrease* operator to eliminate a set of elements from a Partial CISP ("pipelineCISP") starting at a specific element, namely the "Gaussian" element.

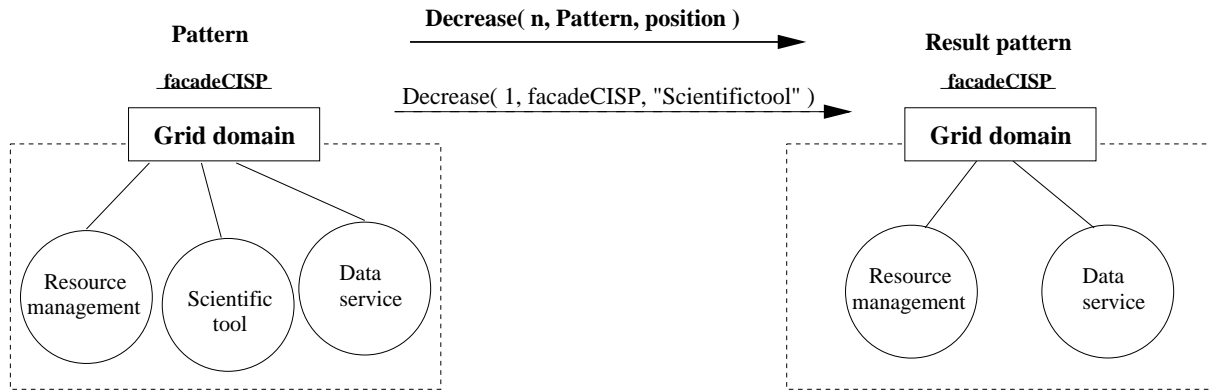


Figure 5.15: Elimination of one particular element of the "facadeCISP", namely "Scientific-tool".

be removed upon invocation of the *Decrease*, the argument "n" should have the value one, i.e. $\text{Decrease}(1, \text{pipelineCISP}, \text{"Gaussian"})$. Likewise, the removal of a specific instantiated CPH from the "facadeCISP" pattern would require the value one for the parameter "n", e.g. $\text{Decrease}(1, \text{"facadeCISP"}, \text{"Scientific tool"})$ as represented in Figure 5.15.

Please note that, it is not allowed to use the $\text{Decrease}(n, P, \text{position})$ operator version to remove a crucial element of a non-topologic CISP or one element that disrupts the structural semantics of the pattern. For example, it is not possible possible to delete the "Grid domain" element from the "facadeCISP" in Figure 5.14, as it is not possible to delete the nucleus of a *Star* CISP, nor the "realsubject" of a *Proxy* CISP.

Next section concludes the description of the possible actions in the fourth methodology step defined in section 5.2.1, namely the operation of Structural Patterns combined with (one or more) Behavioural Patterns (i.e. *SB-PTs*) but whose elements are already bound to specific executables, i.e. *Pattern Instances (PIs)*.

5.2.4 Operating Pattern Instances (PIs)

In this section we clarify the operation of a *Pattern Instance (PI)* which represents:

- a fully or partially *Component Instantiated Structural Pattern (CISP)* (defined previously in section 5.2.3) annotated with one or more Behavioural Patterns;
- the result of associating (all or a sub-set of) the component place-holders of a *SB-PT* (a Structural Pattern Template combined with one or more Behavioural Pattern Templates, as defined in section 5.2.2) to executables representing components/services/tools.

Moreover, we recall the distinction between *Regular SB-PTs* and *Heterogeneous SB-PTs* defined in section 5.2.2 and exemplified in section 5.2.2. Such differentiation is also made in the context of PIs, namely:

- A *Regular PI* represents a Pattern Instance (PI) whose present and future elements are to be coordinated by a Single Behavioural Pattern at run-time. Specifically, the data and control flows annotations associated to pre-existing and future elements are well defined, both at development time and at run-time. Therefore, the user does not need to explicitly define the flow dependencies for the new added elements since their behaviour is established by default.
- An *Heterogeneous PI* represents the combination of a single (partially or totally) instantiated Structural Pattern with more than one Behavioural Pattern providing the pattern's elements with different behaviours. Consequently, new elements of the *Heterogeneous PI* are not automatically associated with pre-defined data and control flow dependencies, but these have to be explicitly defined by the user.

Furthermore, we also define the possibility of operating PIs as *first-class entities*, similarly to what was described for SB-PTs. Consequently, these PIs, designated as *FC-PIs*, can also be, for example, replicated or replaced as a single entity, but their underlying structure and behavioural annotations are also directly accessible towards independent structural and behavioural reconfiguration purposes.

Finally, and concerning all possible operations over PIs, these are identical to the three ones described for *SB-PTs* in section 5.2.2, but with the necessary distinction between development time and run-time. Namely, on one hand, *Ownership* operations, *Structural* operations, and the *modification of behavioural annotations* are all applicable to PIs at development time, and with similar results to what was explained for SB-PTs. On the other hand, those three types of pattern manipulation are also possible at run-time but, in some cases, under the control of the other possible operations, namely, *Execution Operators* (previously presented in sections 3.3.5 and 4.4).

The description of overall run-time manipulation of PIs is deferred to section 5.3, whereas the following two sub-sections describe, respectively, *Structural operations* and the *modification of behavioural annotations* over the different types of PIs at development time. As for *Ownership operations* at development time, these are similar to operating S-PTs or SB-PTs, and are, again, not discussed.

I – Structural Operation of PIs

First of all, the application of the *Eliminate* operator to a PI results in its deletion – any annotations related to behaviour or to the binding to executables are completely discarded.

Moreover, and similarly to CISPs, the *Reshape* operator cannot be applied to any *Pattern Instances (PIs)*. A PI with a particular behaviour cannot therefore be transformed into another PI comprising a different structure (and that would either present the same behaviour or a different one).

The *Replace* operator, in turn, can only be applied to a *First-Class Pattern Instance (FC-PI)*, and it may be substituted for another FC-PI, for a S-PT, or for a SB-PT. For these two latter replacing patterns, the user has to subsequently apply, as necessary, additional behavioural annotations and to instantiate the component place-holders to executables in order to build a new PI.

As for the *Replicate* operator, in case the PI is manipulated as a FC-PI, a new complete and identical PI is created. However, the Structural Pattern within the PI is also directly accessible, meaning that in this case the *Replicate* operation results in the creation of a new S-PT with the same structure as the operated PI.

The *Group* and *Ungroup* operators, in turn, handle PIs as FC-PIs: the grouped FC-IPs are represented by the aggregate as a whole, and at any time the aggregate can be dissolved through the *Ungroup* operator but the inner PIs remain existing.

Clearly, the *Embed* operator can only embed FC-PIs and into a pattern which has free component place-holders (CPHs), may it be already a *Hierarchical PI*, or not. In fact, and in case of the destination being a *Hierarchical PI*, this can only be either a *group*, or a partially instantiated PI – *Partial PI* (i.e. PI that has at least one free component place-holder). The *Extract* operator can also only remove a FC-PI from within a *Hierarchical PI*, being the latter either a group-based, or a partially / fully instantiated *Hierarchical PI* (i.e. *Partial Hierarchical PI* or *Full Hierarchical PI*).

Finally, and considering the application of the *Increase/Decrease* and *Extend/Reduce* operators, these operators make the distinction between manipulating, or not, a PI as a first class entity. Specifically, the manipulation of FC-PIs results in pre-defined behavioural annotations to be automatically added to the new elements within FC-PIs, optimising therefore a FC-PI's reconfiguration. We discuss first the *Increase/Decrease* operators, with the exception of PIs with a structure conforming to the *Adapter Structural Pattern*, as those operators are not applicable in this situation. Afterwards, this section ends with the *Extend/Reduce* operators.

a) – Applying the Increase Operator to a PI

The manipulation by the *Increase* operator (in its two versions) of a PI results in the creation of “n” new component place-holders, being “n” the number passed as argument to the operator. As expected, the new elements are structurally connected to the other elements in conformity to the Structural Pattern within the PI. Furthermore, the behavioural annotations

of the pre-existent elements remain the same after the *Increase* operation and, by default, the new component-place holders are not tagged with any behavioural annotations. As a result, a structural reconfiguration independent from the behaviour is accomplished. The user may then annotate the new elements with roles within a specific Behavioural Patterns.

However, in the case of a *Regular PI*, as the ones suggested in section 5.2.2, the behaviour of the new elements is predictable, since a single *Behavioural Pattern* is to rule their flow and data dependencies at run-time in a well defined way. For this reason, it is possible to annotate automatically those new elements with pre-defined roles within that Behavioural Pattern. Contrarily, an *Heterogeneous PI* is to be coordinated by two or more Behavioural Patterns, and therefore, it is up to the user to associate new elements with a behaviour.

Nevertheless, we also offer the possibility of not defining automatically the roles of new elements within a *Regular PI*. As such, only if the *Regular PIs* are manipulated as *first class entities* (i.e. a *Regular FC-PI*) the behavioural annotations are added automatically. Otherwise, it is up to the user to annotate explicitly the new elements with specific roles, for example, within a different Behavioural Pattern and, consequently, transforming the *Regular PI* into an *Heterogeneous PI*.

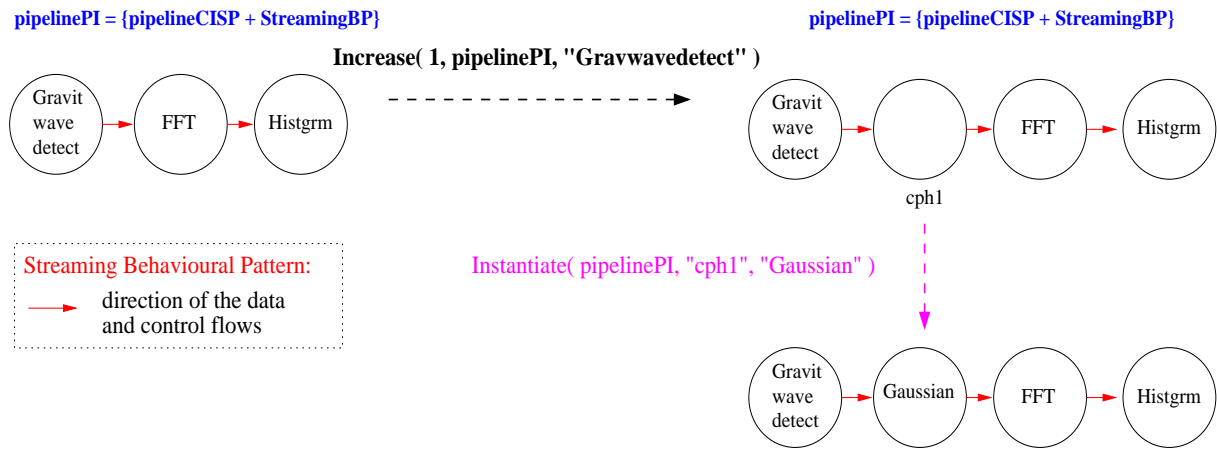


Figure 5.16: Increasing a Regular FC-PI by one element, namely a pipeline pattern combined with the Streaming Behavioural Pattern.

Figure 5.16 presents an example of increasing the number of elements of a *Regular FC-PI*. The PI in the Figure represents a pipeline for gravitational wave processing which is to be coordinated, at run-time, by the *Streaming Behavioural Pattern*. As shown in the left-side of the Figure, the pipeline has three stages defined with data and control flows which will obey the *Streaming* pattern. Namely, the first stage is associated with a tool for detecting gravitational waves ("Gravwavedect") and it is marked to produce data to a Fast Fourier transformation component ("FFT"). This component, in turn, is to send the transformed data to a visualisation tool ("Histgrm").

Since the component place-holders within a PI are already bound to executables, it is necessary to define where the new component place-holder is to be placed. This is accomplished through the extended version of the *Increase operator*, namely *Increase(n, P, position)*. In the case of the example depicted in Figure 5.16, the new component place-holder is created after the "Gravwavedetect", as shown in the right-hand side of that Figure. Moreover, the new element

is to obey to the *Streaming* Behavioural Pattern, and consequently, its data and flow dependencies are automatically annotated, being similar to the annotations of the pre-existent elements. Subsequently, that component place-holder is bound to a "Gaussian" component for data processing, defining the new stage's action in the pipeline for gravitational wave processing. The instantiation process is represented by the *Instantiate(P, position, component)* operator which was previously described in section 5.2.3.

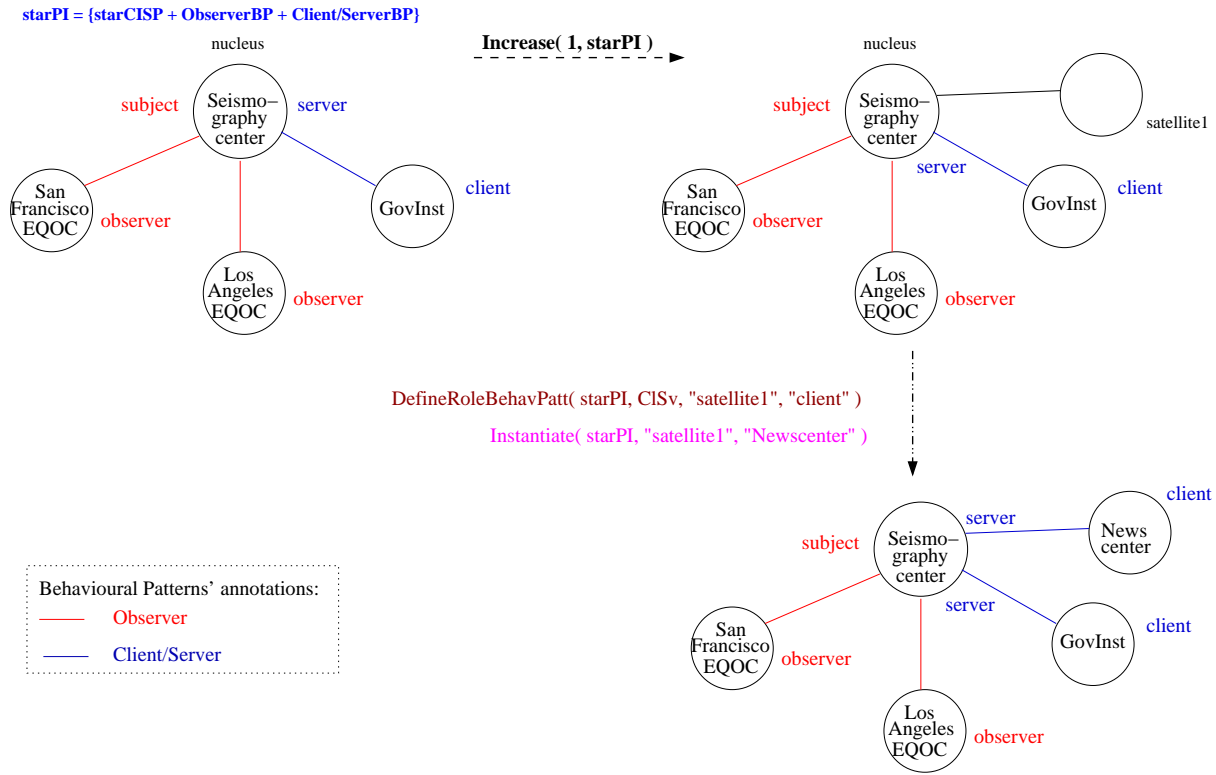


Figure 5.17: Increasing an Heterogeneous PI by one element, namely a star pattern combined with the Observer and Client/Server Behavioural Patterns.

Concerning the modification of an *Heterogeneous Pattern Instance (PI)*, Figure 5.17 depicts a star whose elements are coordinated in different ways. On the left-hand side of the Figure, two satellites within the PI, namely "LosAngelesEQOC" and "SanFranciscoEQOC" (representing two "Earthquake Observation Centers") are referenced as "observers" within the applied *Observer* Behavioural Pattern; the third satellite, namely "GovInst" (representing a "Governmental Institution") is associated with "client" role within the applied *Client/Server* Behavioural Pattern; and finally, the nucleus of the star, "Seismographycenter" in the Figure, is marked for playing two roles – as the "subject" and as the "server" – concerning the two applied Behavioural Patterns.

The addition of an extra element to the star through the *Increase(n, P)* operator version is represented in the right-hand side of Figure 5.17. A new component place-holder, i.e. "satellite1", is created, and it is not automatically tagged with a specific behaviour. The user has to explicitly associate that element with a role, for example, a "client" role. This is displayed in Figure 5.17 by the application of the *DefineRoleBehavPatt(P, B-P, {element, role})* operator presented in section 3.3.6, i.e. *DefineRoleBehavPatt(starPI, ClSv, "satellite1", "client")*.

Finally, the new component place-holder is instantiated to an executable component, i.e. a

"Newscenter" as shown in the Figure, through the *Instantiate*(*starPI*, "satellite1", "Newscenter") operator. After the instantiation, it will be ruled at run time by the *Client/Server* Behavioural Pattern on interacting with the "Seismographycenter".

b) – Applying the Decrease Operator to a PI

The decreasing of the number of elements of a PI is similar to *Component Instantiated Structural Patterns (CISPs)*, but where the behavioural annotations have also to be contemplated. As such, we recall the two definitions for the *Decrease* operator (presented in section 3.3.2 and discussed in section 5.2.3) and define their consequences on the behavioural annotations. Concretely:

Decrease(*n*, *P*) Decrements the number of non instantiated elements, i.e. component place-holders (CPHs) of a *Partial PI* (i.e. partially instantiated PI) "*P*", by the value "*n*". If this number is greater than maximum number of the existing free CPHs, only these CPHs are deleted. Hence, the operator has no effect in the case of a fully instantiated PI (i.e. a *Full PI*). The behavioural annotations associated with the eliminated component place-holders are also removed.

Decrease(*n*, *PI*, *position*) Deletes "*n*" pattern elements (either instantiated or CPHs) from a PI "*P*" starting at, and including, the pattern element identified by the argument "*position*". In case it is not possible to define a structural ordering within the operated PI, the *Decrease* is to be called with argument "1" for the parameter "*n*", meaning that only the pattern element define by "*position*" is removed. The behavioural annotations associated with the removed elements are also eliminated.

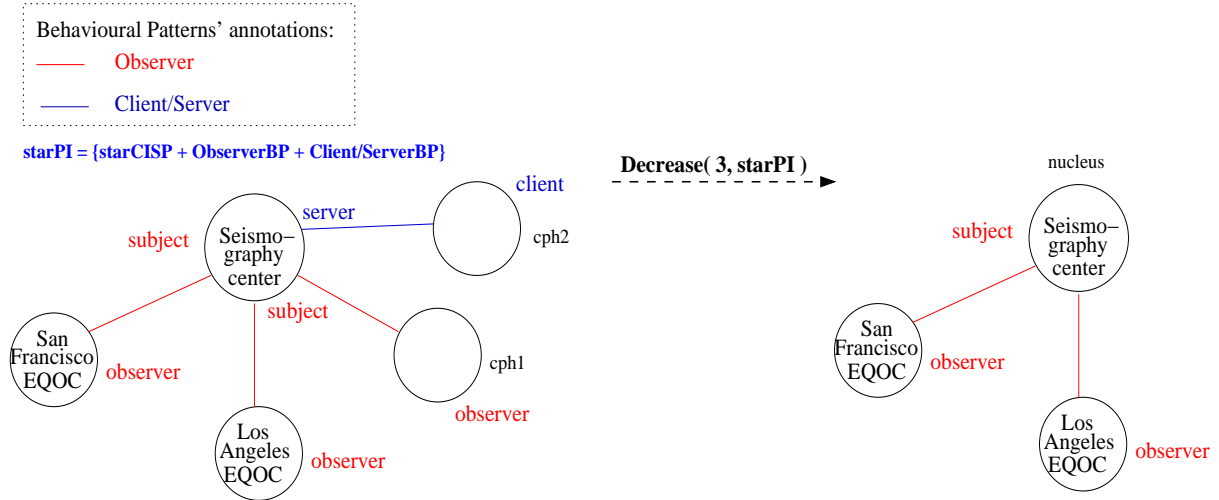


Figure 5.18: Decreasing a partially instantiated Heterogeneous PI by two component place-holders. Although it was requested the deletion of three CPHs, only the two existing CPHs are deleted. All behavioural annotations pertaining to those components are also eliminated.

On one hand, an application example of the first form of the *Decrease* operator, is depicted in Figure 5.18. Specifically, the *Decrease*(3, *starPI*) operator is applied to the *Heterogeneous PI* named "starPI" for the deletion of three component place-holders. Since only two CPHs exist within "starPI", they are eliminated along with their behavioural annotations. The resulting PI

is shown on the right-hand side of the Figure. As can be observed, the “server” role annotation (pertaining to the *Client/Server* Behavioural Pattern) of the nucleus of the star (i.e. “Seimog-raphycenter”) no longer exists, as a result of the removal of the related “client” component place-holder (i.e. “cph2”).

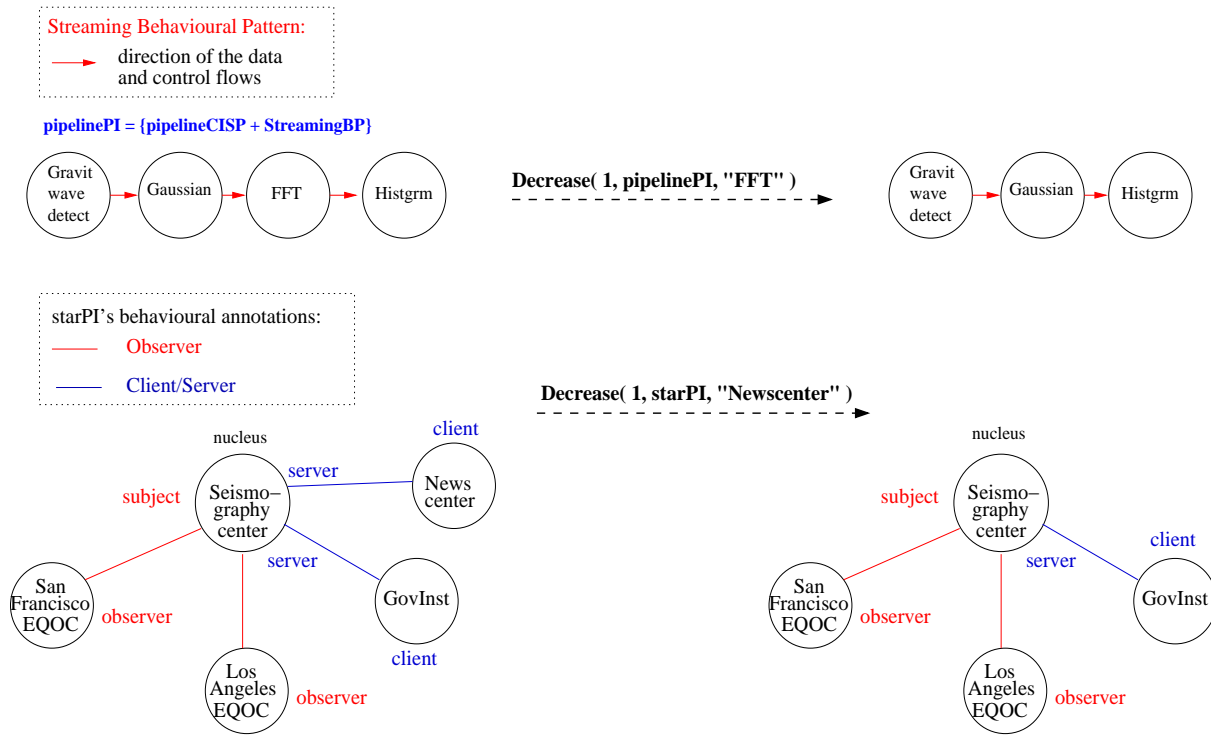


Figure 5.19: Decreasing two PIs by one element. On top, the element “FFT” is removed from the Regular PI “pipelinePI”. On bottom, the element “Newscenter” is removed from the Heterogeneous PI “starPI”.

On the other hand, two application examples of the second form of the *Decrease* operator, i.e. *Decrease(n, P, position)*, are shown in Figure 5.19. The first example is depicted in the upper part of the Figure, where the *Decrease* operator is used to remove one specific element, i.e. “FFT”, from a *Regular PI* (“pipelinePI”). Consequently, the structural connections and the behavioural annotations remain consistent for the other elements.

The second example shows the removal of a particular element from an *Heterogeneous PI*. Concretely, the “Newscenter” element is eliminated from the “starPI”, and all related behavioural annotations are also removed, with no implications on the (still) necessary behavioural annotations.

Finally, two remarks are due:

1. For both versions of the *Decrease* operator, and for particular cases of *Heterogeneous PIs*, i.e. whose behavioural dependencies are defined by two distinct Behavioural Patterns, it may be necessary to check and correct the behavioural consistency of the resulting PI. For example, the deletion of an inner member of a *Pipeline PI* that is annotated with both *server* and *producer* roles (within the *Client/Server* and *Producer/Consumer* patterns, respectively) produces an inconsistent behaviour that has to be corrected by the user.

2. As previously remarked for CISPs, it is not allowed to use the *Decrease*(*n*, *P*, *position*) operator version to remove a crucial element of a PI, i.e. an element whose deletion would disrupt the pattern's structural semantics. For example, it is not possible to delete the “Seismography center” element from the “starPI” in Figure 5.19, since it is not possible to have a *Star* pattern without a “nucleus” element within the structure.

c) – Applying the Extend and Reduce operators to a PI

Considering the manipulation of PIs by the *Extend* and *Reduce* operators, such is similar to what was described for SB-PTs in section 5.2.2 with the difference that now the pattern argument is a SB-PT whose all or a sub-set of its component place-holders are already bound to executables.

Augmenting a PI through *Extend* results in the addition of an element in conformity to the Structural Pattern within the PI. Equivalently to SB-PTs, the *Extend* operator can either: a) act over the Structural Pattern underlying the operated PI; or b) manipulate the PI as a *first class entity* (FC-PI). In the first situation, the structure is extended in conformity to the operated Structural Pattern and independently from the applied behavioural annotations. This operation is similar to what was specified in section 5.2.3 for *Component Instantiated Structural Patterns* (CISPs). In the second situation (i.e. “b”), the behavioural annotations are taken into consideration if the PI is *Regular*, analogously to what was defined for *Regular FCSB-PT* in section 5.2.2. Therefore, the new added element, resulting from the *Extend* operation, is automatically annotated with a role within the applied Behavioural Pattern.

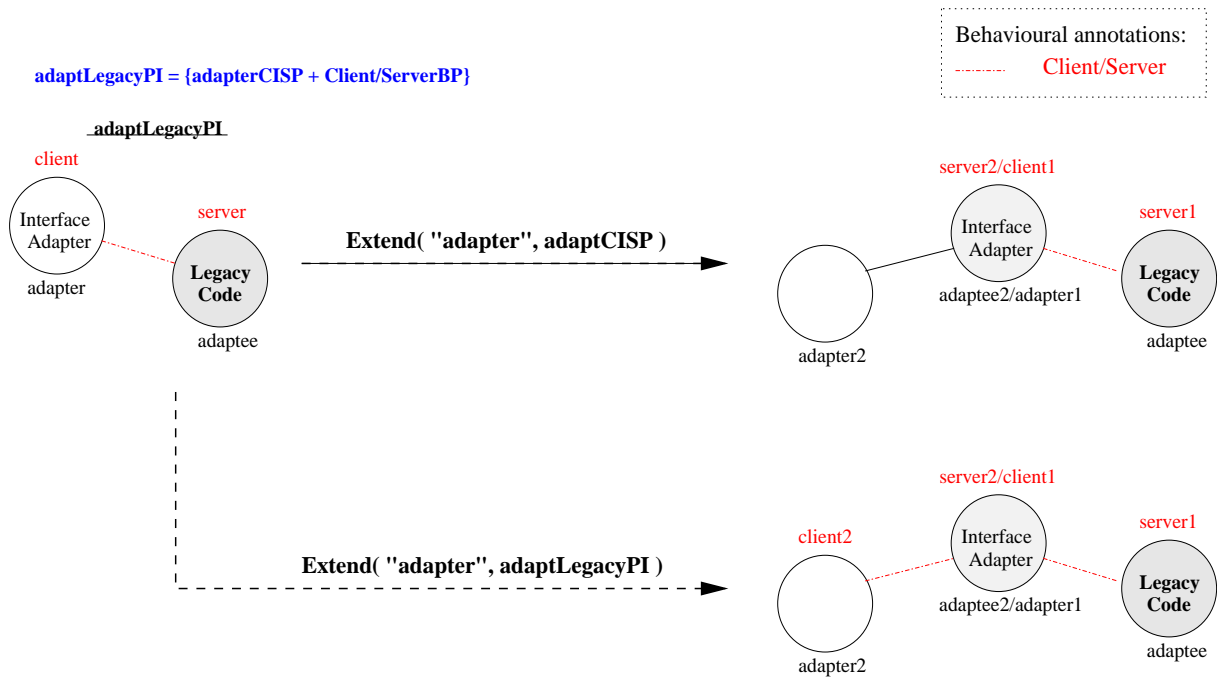


Figure 5.20: Applying the Extend operator to a PI (“adaptLegacyPI”). At the top, the structure is augmented disregarding the applied Behavioural Pattern. At the bottom, the PI is operated as a Regular FCSB-PT which results in the automatic annotation of the new element with a role within the applied Behavioural Pattern.

Figure 5.20 depicts an example of the above two situations on extending a PI. The underlying structure of the operated PI is an *Adapter* Structural Pattern, and the data and flow

dependencies are to be ruled by the *Client/Server* Behavioural Pattern. The PI, shown on the left hand side of the Figure, is named “adaptLegacyPI” and represents the adaptation of the interface of a legacy code (“LegacyCode”) that supports some service. The “LegacyCode” is the “adaptee” within the semantics of the *Adapter* SP and it is annotated with the “server” role, and the “InterfaceAdapter” instantiates the “adapter” within that semantics and it is annotated as the “client”.

The result of extending the “adaptLegacyPI” independently from its behavioural annotations is represented on the top of the right-hand side of Figure 5.20 – a new adapter is created (“adapter2”) and no behavioural annotations are attached to it. In turn, the bottom of the Figure illustrates the result of augmenting the “adaptLegacyPI” as a *Regular FC-PI* – the new “adapter2” element is automatically annotated with the “client” role within the applied Behavioural Pattern.

Conversely, the application of the *Reduce* operator to a PI does not need to distinguish between operating only the structure of a PI or operating it as *FC-PI*, since the elimination of the necessary elements implicates the removal of any associated behavioural annotations. Therefore, a PI operated by *Reduce* obeys the structural semantics of the underlying Structural Pattern, which were exemplified in Figures 4.29 and 4.30 in section 4.3, and it is similar to what was defined for *SB-PTs* in section 5.2.2.

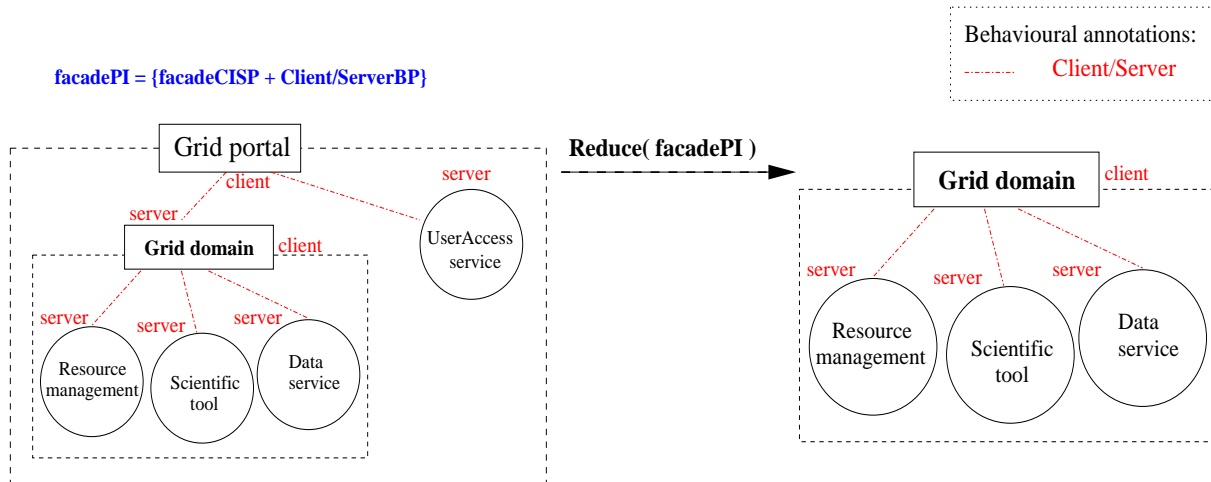


Figure 5.21: Applying the *Reduce* operator to a PI.

Figure 5.21 presents a case of applying the *Reduce* operator to a PI named “facadePI” which consists of a *Facade CISP* whose elements are annotated according to the *Client/Server* Behavioural Pattern. As a result, the outmost Facade S-P is eliminated jointly with the subsystem (“UserAccessService”) that resulted from a previous *Increase* operation. The outmost facade element is now “Grid domain” whose behavioural annotation as a “server” to the eliminated structural element (i.e. the facade “Grid portal”) is also removed since it is no longer necessary.

Next sub-section discusses how to change the behavioural annotations within a PI.

II – Behavioural Modification of PIs

The modification of the behavioural annotations within a PI is identical to what was defined for SB-PTs in section 5.2.2. Given a *Regular PI*, its applied Behavioural Pattern can be replaced as a whole for another Behavioural Pattern that also results into another *Regular PI*. The *ReplaceBehavPatt*(*SB-P*, *B-P1*, *B-P2*) operator defined in section 3.3.6 supports such behavioural change independently from the underlying structure, with the automatic replacement of the behavioural annotations onto all elements within the PI.

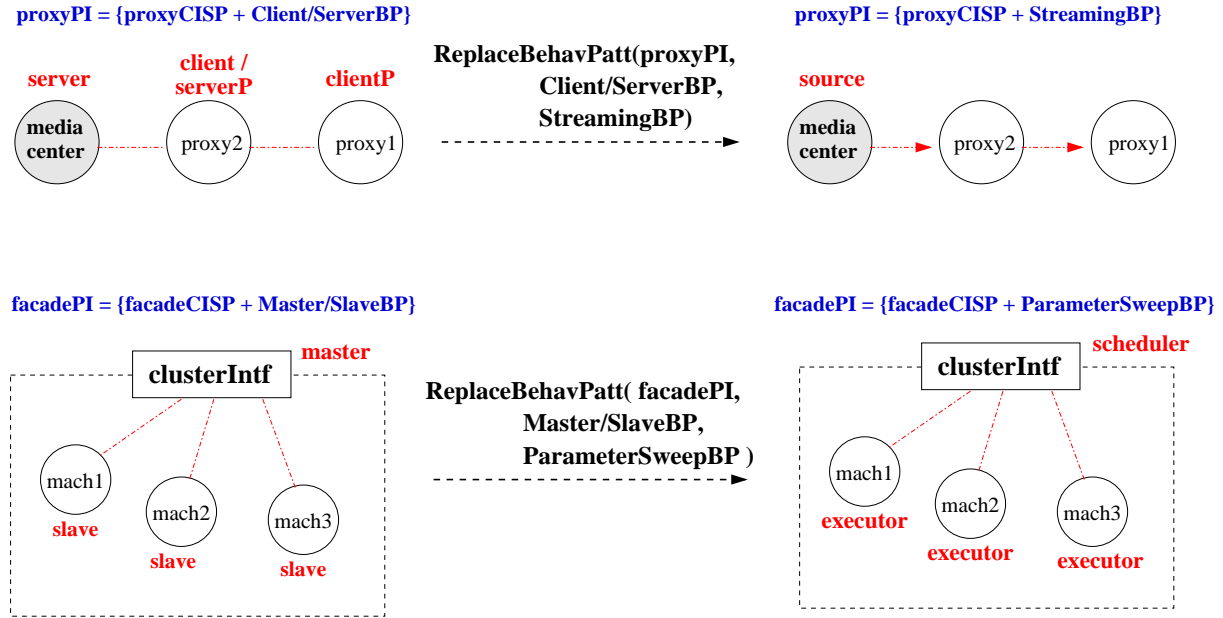


Figure 5.22: Changing the behavioural annotations of two Regular PIs.

Figure 5.22 presents two examples on replacing the behavioural annotations within two PIs. On top of the Figure, the behaviour associated to a proxy supporting the remote access to a media center (“proxyPI”) is changed from the *Client/Server* pattern (left-hand side of the Figure) to a *Streaming* pattern (right-hand side of the Figure). The *Client/Server* behaviour supports media download, whereas a switch to the *Streaming* pattern represents the “media center” now acting as remotely accessible “streaming media system”.

The bottom of Figure 5.22 displays a facade representing the interface to a machine cluster (“facadePI”). On the left-hand side of the Figure, the behaviour is to be controlled by the *Master/Slaver* pattern supporting the parallel execution of particular applications. However, this behaviour may be switched to the *Parameter-Sweep* pattern in case of specific applications that benefit from this pattern (e.g. Monte-Carlo simulations).

Next section discusses the different ways to reconfigure an application which is already being executed.

5.3 Reconfiguration

One major characteristic to be provided to application/system designers (either experts or non-experts) is the possibility to adapt a defined configuration to both respond to new user require-

ments and to incorporate new system capabilities. Preferentially, such adaptation should also be possible at runtime. Therefore, we highlight in this section the potentiality of our model on supporting both static and dynamic reconfiguration. Namely, the structural and behavioural operation of patterns as first class entities allows the structural change of a configuration independently from its behaviour, and vice-versa. Moreover, *Full PIs* (i.e. *Full CISP*s combined with one or more Behavioural Patterns) may be replaced with other Pattern Instances.

Whereas the above changes may be done at development time (as previously described in section 5.2), the model may also support some degree of run-time reconfiguration. The first sub-section suggests the possible ways to reconfigure a running pattern-based application, and the second presents a few examples.

5.3.1 Reconfiguration Options

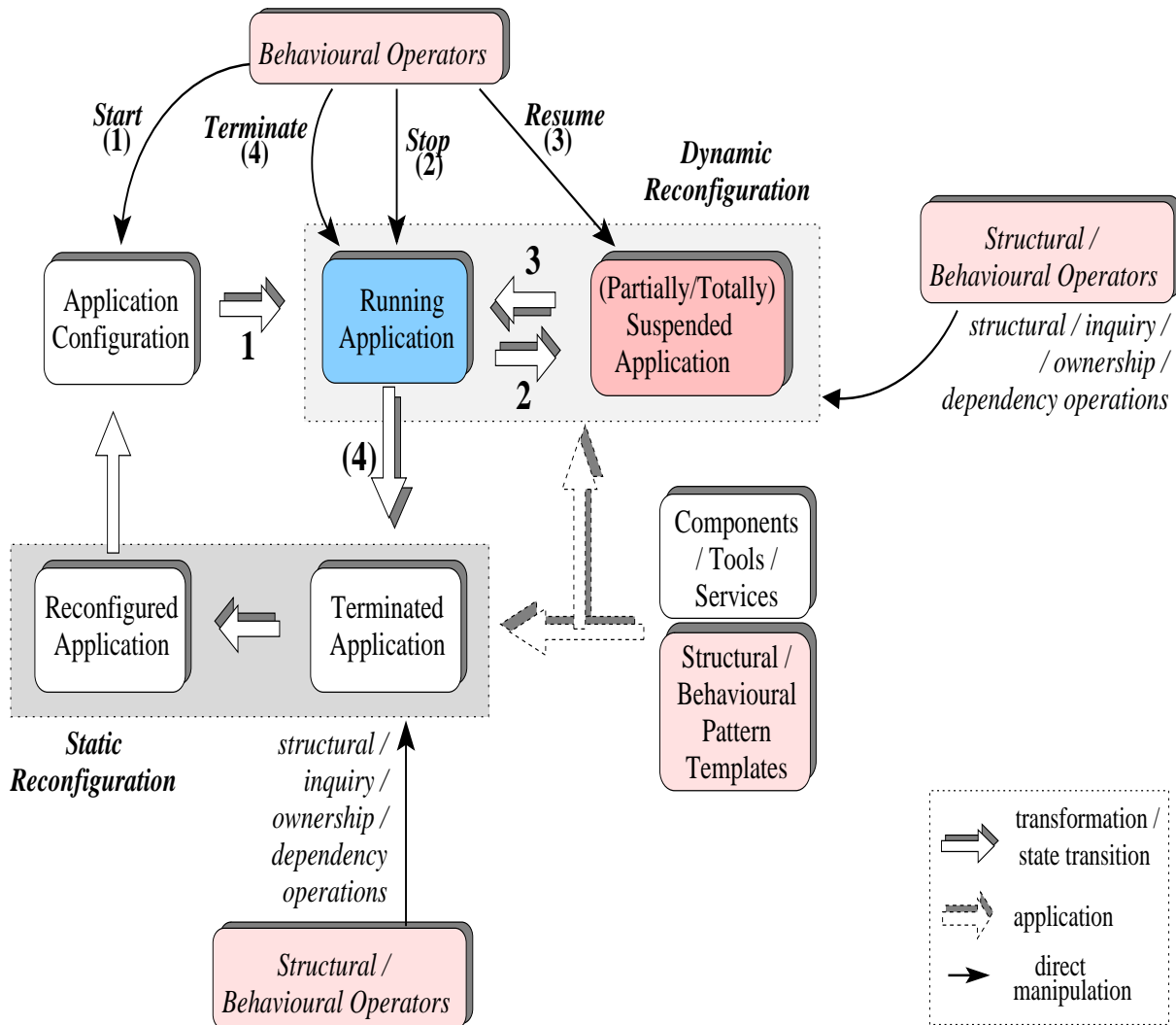


Figure 5.23: Summary of the possible steps for reconfiguring a running application.

There are three ways to reconfigure a running (pattern-based) application:

A to abort its execution, apply the necessary modifications, and re-execute it;

- B to suspend the application's execution, either completely or partially, and apply the desired changes;
- C to operate the running application, while trying to reduce the perturbation upon its execution.

The first situation corresponds to a *Static Reconfiguration*, whereas the other two may be classified as *Dynamic Reconfigurations*. These situations are identified in Figure 5.23 which aims to represent a general view towards reconfiguring pattern-based applications.

Before explaining the above reconfiguration options, it is worthwhile recalling the role of the *Start*, *Stop*, *Resume*, and *Terminate* Behavioural Operators on controlling *state transitions*. Similarly to what was described in section 4.5 for a single *Pattern Instance*, the transitions between different execution states for an *Application Configuration* are also controlled by the above operators. Specifically:

- The *Start* operator causes the application to switch to the "running" state (transition 1 in Figure 5.23).
- The *Stop* operator originates a shift from the "running" to the "suspended" state (transition 2). Since this operator may be applied to individual *Pattern instances* within the application, the application may be only partially suspended.
- The *Resume* operator changes the application back to "running" state (transition 3). According to what is represented in Figure 5.23, the transition from a (*Partially/Totally*) *Suspended Application* to a *Running Application* only occurs when all previously suspended *Pattern Instances* are switched to the "running" state.
- The *Terminate* operator aborts the execution of the *Running Application*, generating a transition to *Terminated Application* (transition 4).

These operators play a major role in the first and second reconfiguration options enumerated above. Namely, a *Static Reconfiguration* begins with the application of the *Terminate* operator as shown in Figure 5.23, followed by the application and operation of Structural and Behavioural PTs and operators, and the instantiation of the component place-holders to executable components (or services/tools). Subsequently, the resulting reconfigured application may be (re-)launched through the *Start* operator.

The second reconfiguration option enumerated above concerns the modification of a running application but whose transformations can only be applied if the execution of the entire application or of sub-parts of it are suspended. Therefore, this kind of *Dynamic Reconfiguration* requires the application of the *Stop* operator to suspend the execution of the particular running *Pattern Instances* (or the entire application) upon which changes have to be applied to. After the employment and operation of Structural and Behavioural PTs and the instantiation of the component place-holders to executable components (or services/tools), the *Resume* operator is used to proceed with the normal execution, as depicted in Figure 5.23.

This way to reconfigure a running application, namely, by acting upon individual PIs, permits changing parts of the application, but at a higher granularity than the component level (e.g. PIs may be replaced as a single entity, or structural/behavioural adjustments may be

restricted to individual PIs). Moreover, the manipulated PIs may represent independent sub-systems whose individual reconfiguration may have a reduced impact upon the overall application's execution.

The third reconfiguration option, namely, the modification of a running application without suspending its execution, is also based on acting over individual PIs.

The next sub-section presents a few possible examples on how to modify a running application concerning the second and third reconfiguration options enumerated above. As for static reconfiguration (first reconfiguration option above), it was already discussed and examples were presented throughout section 5.2.

5.3.2 Reconfiguration Examples

As described in the previous section, one way to reconfigure a running application is to act upon it and trying a minimal disturbing of its execution. Taking the characteristics of our model into consideration, such modification is only possible in a restricted number of cases. On one hand, we define as possible, the dynamic reconfiguration (i.e. with no need for stopping the application's execution) of the structure of *Regular PIs*. In fact, the addition/elimination of elements does not disrupt the overall execution, and the behaviour of the new added elements is pre-defined. The first two examples aim to illustrate such situation. On the other hand, it is also possible to reconfigure a partially or totally suspended application (i.e. as a result of the *Stop* operator). The third example illustrates the latter case.

First Example

Figure 5.24 presents an example that consists of a *Regular* Pattern Instance based on the Star Structural Pattern whose elements' data and flow dependencies are ruled by a single Behavioural Pattern, namely, the Client/Server. As shown in the Figure, the number of satellites is augmented by one, firstly by adding a new satellite component place-holder to the structure, and secondly, by instantiating it with a runnable component whose execution is automatically launched. Such procedure is represented in the Figure by the *InstantiateRunnable(gridservicePI, "satellite1", "Gridclient4")* action that corresponds to the implementation mechanism of associating the selected executable to "satellite1" and launching its execution. As it would be expected, the new element's dependencies to the nucleus of the Star, namely "Gridserver", are to be ruled by the Client/Server pattern, similarly to what happens to the other satellites.

Second Example

Figure 5.25 presents a dynamic reconfiguration as a way to build a dynamic itinerary for a mobile agent. This may be useful, for example, for a "Grid agent" whose visited "grid services" are to be dynamically defined. The PI representing the "Grid Agent" relies on the *Proxy* Structural Pattern combined with the *Itinerary/Mobile Agent* Behavioural Pattern. Such combination defines a *Regular PI*, and the agent is moved simply by operating the PI through *Extend*. A chain of proxies is left behind, allowing the remote access to the "Grid Agent" wherever its location.

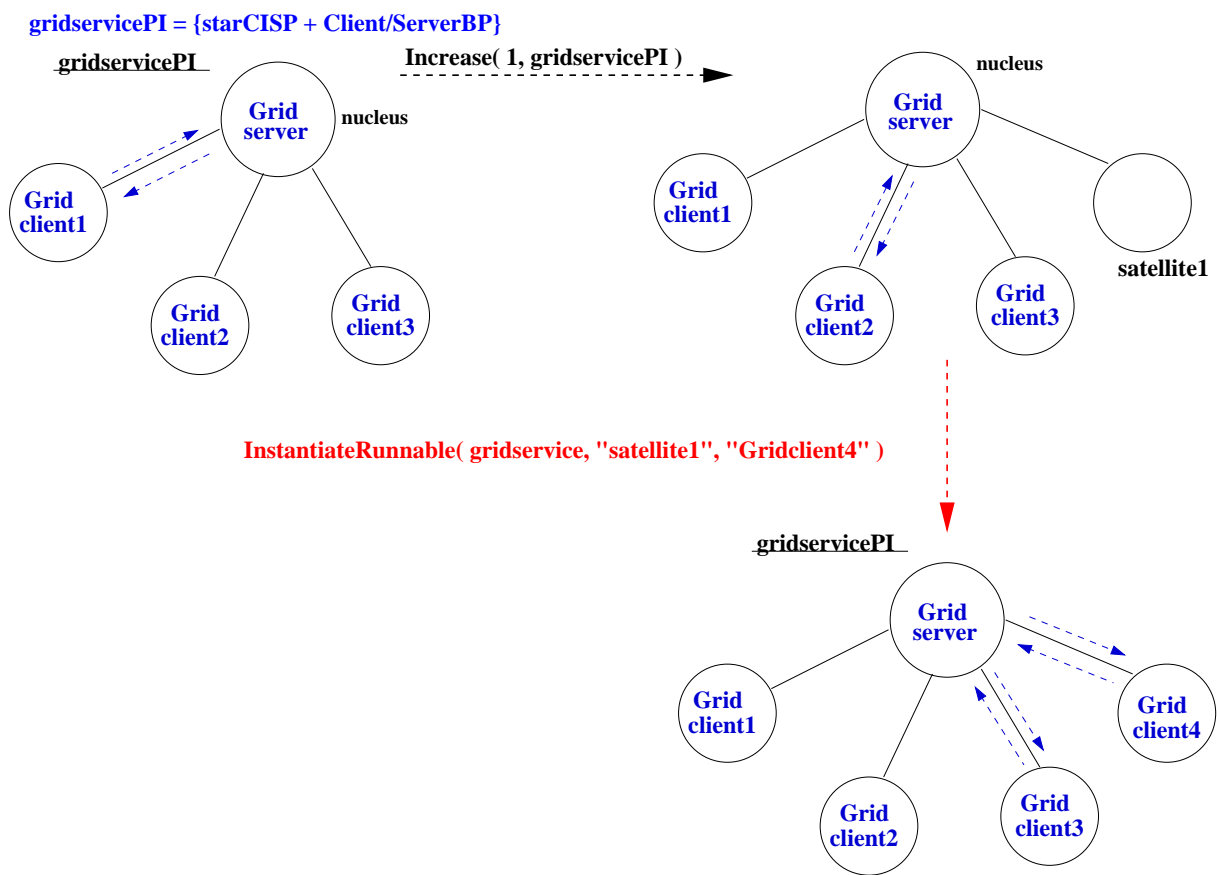


Figure 5.24: The dynamic reconfiguration of a Regular Pattern Instance representing a Grid service.

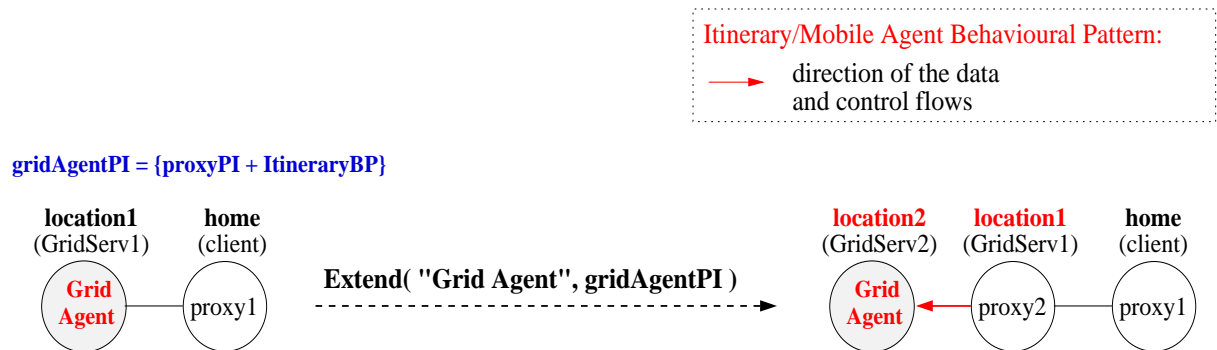


Figure 5.25: Building a dynamic itinerary for an Agent

Third Example

The other way to reconfigure a running application, is to suspend its execution temporarily (through the *Stop* operator), operate its configuration through the available Structural and Behavioural operators, and then resume execution within the new configuration.

Figure 5.26 presents an example of a running application which has to be stopped in order to be reconfigured. Concretely, the pre-defined itinerary (i.e. defined at development time) for a “Grid Agent” has to be changed. In order to guarantee that the agent does not miss an intermediate new stage within the Itinerary, the entire PI is to be suspended through the *Stop*

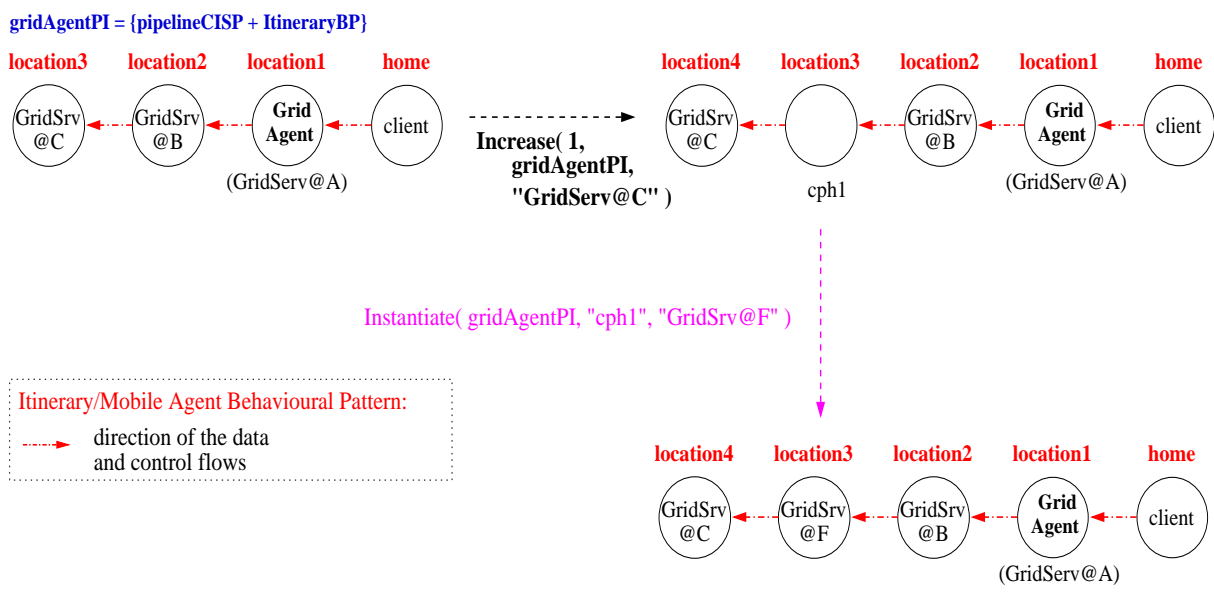


Figure 5.26: *Reconfiguring a Pattern Instance whose execution needs to be stopped.*

operator. The left-hand side of Figure 5.26 presents the already stopped PI that supports the pre-defined Itinerary for a “Grid Agent” – the PI is named “gridAgentPI”.

The inclusion of a new (intermediate) destination within the itinerary is achieved by operating “gridAgentPI” through *Increase*. This operator allows the definition of the specific position where to insert the new element, namely, after “GridSrv@C” (which represents a Grid service at location “C”). Subsequently, the new element is instantiated with the new destination, namely, “GridSrv@F” (a Grid Service provided at location “F”). The execution of “gridAgentPI” can thereafter be resumed with the guarantee that the new location is also visited.

5.4 Summary

This chapter described reconfiguration strategies for applications built as a result of pattern composition, both for development time and also while an application is already executing. To that extent, the chapter also discussed an extended version of the methodology associated to the model which defines pattern manipulation at the different phases of the application development cycle.

Namely, we proposed an approach for control of the reconfiguration process itself, namely, reconfiguration may be done explicitly by the user, on a per pattern basis, and the reconfiguration steps may be defined as sequence of operators applied to those patterns. Furthermore, individual pattern reconfiguration may be performed in the two dimensions of structure and behaviour, either independently or jointly.

Nevertheless, the reconfiguration capabilities discussed in this chapter still have to be fully validated in terms of their adequacy to solve the problems inherent to dynamic reconfiguration previously mentioned in Chapter 2, and also raise other problems related with the coordination of the behaviours in a Hierarchical Pattern. Such was not possible to be studied in the context of this thesis, but will be the subject of future research concerning our model.

6

The Architecture and its Implementation

Contents

6.1	Introduction	170
6.2	The Architecture Supporting the Model	170
6.3	An Instance of the Architecture: Implementation over Triana	177
6.4	Patterns and Operators in Triana	185
6.5	Mapping to the DRMAA API	208
6.6	Summary	216

This chapter describes an architecture supporting the model and a specific implementation of a subset of the model based on an extension of the Triana workflow system. This chapter also discusses the partial mapping of the architecture onto a distributed programming interface, namely the DRMAA API.

6.1 Introduction

To provide the user with the possibility of configuring and controlling the execution of a Pattern- and Operator-based application, these have to be integrated in an environment that offers adequate support for all stages of the cycle application development process.

At design time, patterns are meant to define common inter-relations among state-of-the-art abstractions involving components and services towards reuse. Therefore, patterns have to be provided as first class-entities similarly to components/services so that they are manipulable entities in the configuration process. Moreover, adequate support has to be provided for Structural Operators for their effective usage on pattern refinement.

Additionally, at run-time, Patterns have to remain as operable entities, and Behavioural Operators, in particular, require a distributed execution environment with control capacities over application execution.

In order to fulfill the above requirements, we selected the Triana Problem Solving Environment [20] as the host implementation platform. Triana is a component-based workflow tool that provides support for different kinds of distributed execution, including Grid access.

Hence, this chapter aims to clarify the implementation of the main concepts in the proposed approach within Triana. Specifically, the first section describes a generic layered architecture suitable to support the realisation of Patterns and Operators. The specific implementation over Triana is described in the subsequent sections. Moreover, and due to the relevance of standard generic Distributed Resource Managers Interfaces to control the execution of several resource managers, the last section sketches a possible mapping of some of the Behavioural Operators to the DRMAA API [43].

6.2 The Architecture Supporting the Model

Figure 6.1 represents a view of the generic architecture supporting the model based on patterns and operators. Two major *Layers* shown as boxes may be identified:

- the upper layer, i.e. the upper bigger box in the Figure named *Layer 1*, describes the phases of application design and mapping of components into a generic distributed platform;
- the second layer, i.e. the lower bigger box in the Figure named *Layer 2*, represents the necessary entities for the resource allocation, activation and distributed execution control of the application's components.

On the upper layer (*Layer 1*), the architecture is represented by three *levels*:

- The first level, i.e. *Level 1*, represents a composition environment which supports the interface with the user. Examples are Problem Solving Environments and Portals, which allow the selection of components from the second level, i.e. *Level 2*, and their interconnection for application structuring. In general, the first level provides the user with an integrated environment for the development of a class of applications.

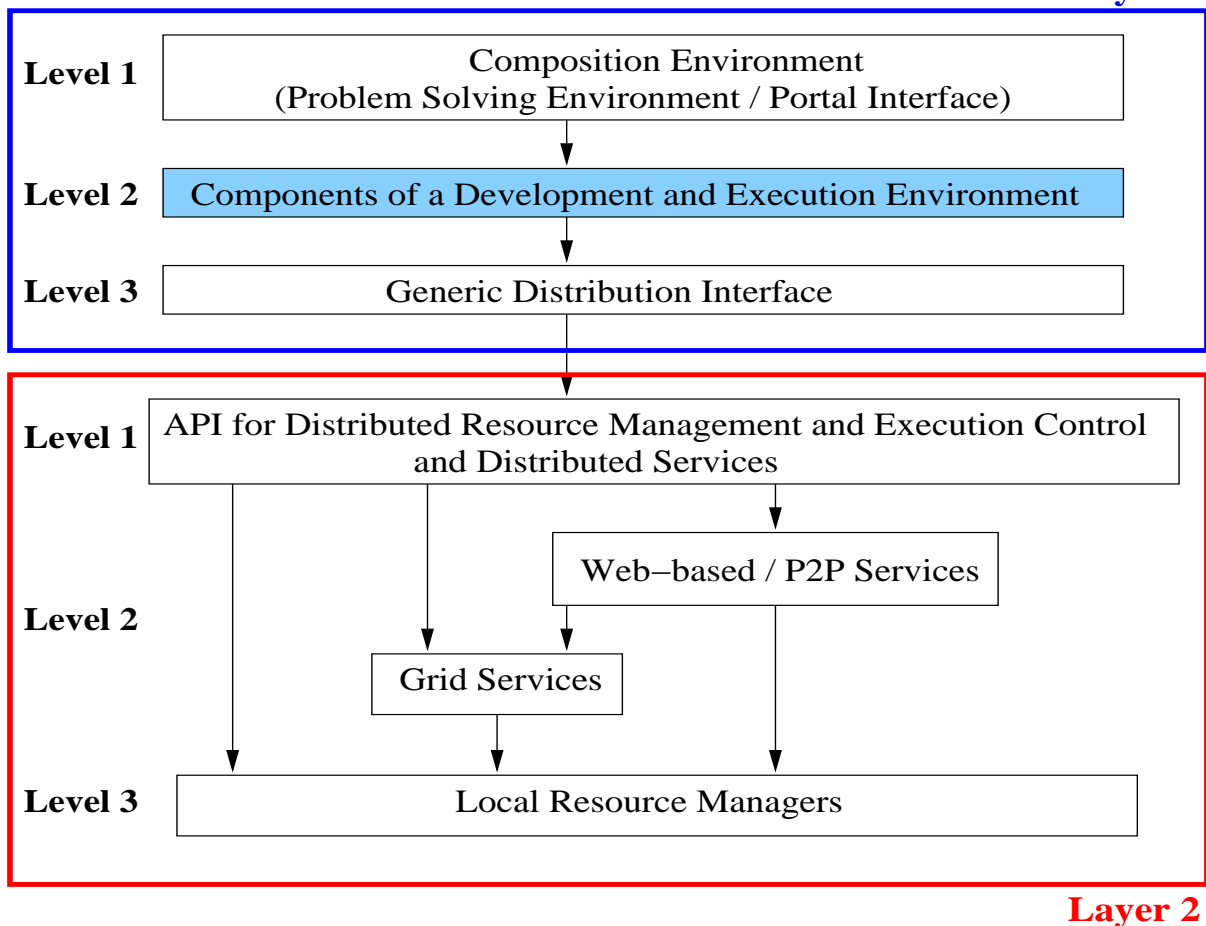


Figure 6.1: A generic architecture that supports the model based on Patterns and Operators.

- The components represented by the second level, i.e. *Level 2*, may be, for example, simulators, visualisation tools, monitoring and steering tools, or coordination components, which are relevant for application configuration in a particular area.
- The third level, i.e. *Level 3*, represents a *Generic Distribution Interface* which provides capabilities for explicitly running and controlling the execution of components in a specific remote place. However, the distribution capability may be completely transparent to the user. The *Composition Environment* (*Level 1*) uses the features of this third level to provide a distributed execution environment to the user. For example, the particularities associated with different kinds of “Grids” infrastructures are hidden by the *Generic Distribution Interface*.

At the lower layer (*Layer 2*), several entities provide the effective distribution and execution capabilities supporting the running application.

- The first level (*Level 1*), i.e. *API for Distributed Resource Management and Execution Control and Distributed Services*, hides the programming specifics of the execution system underneath. This first level provides simpler interfaces for distributed resource management which allow
 - the allocation of the necessary resources for the application and the control of the distributed (execution) among the local resource managers;

- access to Grid interfaces like COG or the OGSA standard which avoid programming directly over the *Globus* [71] system;
 - or access to Web and Peer-to-Peer services.
- The lower levels (*Levels 2 and 3*) represent the low-level distributed services:
 - Grid services which give support to the distributed execution over several heterogeneous resource managers across different organisation boundaries;
 - Web services which provide different functionalities accessible through a standard interface, and may bridge the access to specific services like Grid services;
 - and Peer-to-Peer services which provide a flexible and scalable way for execution distribution.

The implementation of Patterns and Operators relies on *Layer 1* defined in Figure 6.1. Namely, they are integrated as entities in a *Composition Environment* (*Levels 1 and 2* in *Layer 1* in the Figure) and benefit from a *Generic Distributed Interface* (*Level 3* in *Layer 1*). In order to support all the described capabilities of our model, such interface has to provide adequate mechanisms for resource management and distributed execution control and depend on obtaining monitoring information. One example of such desired capabilities is to suspend the execution of all distributed components in a pattern, e.g. executing in a Grid environment, and making a checkpoint of their state; and subsequently resuming the pattern's execution from that saved state. As a result of the complexity of these kind of actions, the mapping between the operators and the particular functionality of a resource management system therefore cannot be pre-defined. We therefore rely on that intermediate API (i.e. the *Generic Distributed Interface* in *Level 3/Layer 1*) implementing a "Super-Scheduler" interfacing local Schedulers necessary to reserve and allocate resources in the Grid (e.g. [5]).

Due to the complexity of our model, only a small subset of the proposed capabilities were effectively implemented. This is described in section 6.3. Meanwhile, next sub-section presents a simple example aiming to highlight how pattern and operator usage relate to the layer that supports the distributed execution of all components in the example.

6.2.1 Application Configuration and Execution Control

The application of Patterns and Operators for structuring distributed applications on Grid-aware environments may be represented by four steps as previously defined in the basic methodology in section 3.1.3. Namely, in the first step, the user selects from a repository the relevant *Structural Patterns* that best configure the application, generating the necessary *Structural Pattern Templates*. In the second step, the user refines the configuration by applying the necessary *Structural Operators*, resulting in a composition of the Structural Patterns. In the third step, the user selects the adequate *Behavioural Patterns* which define the control and data flow dependencies between the elements of the Structural Patterns. Finally, in the fourth step, the user controls the execution of the application by applying the *Behavioural Operators*, which also allow the control of the reconfiguration of the application.

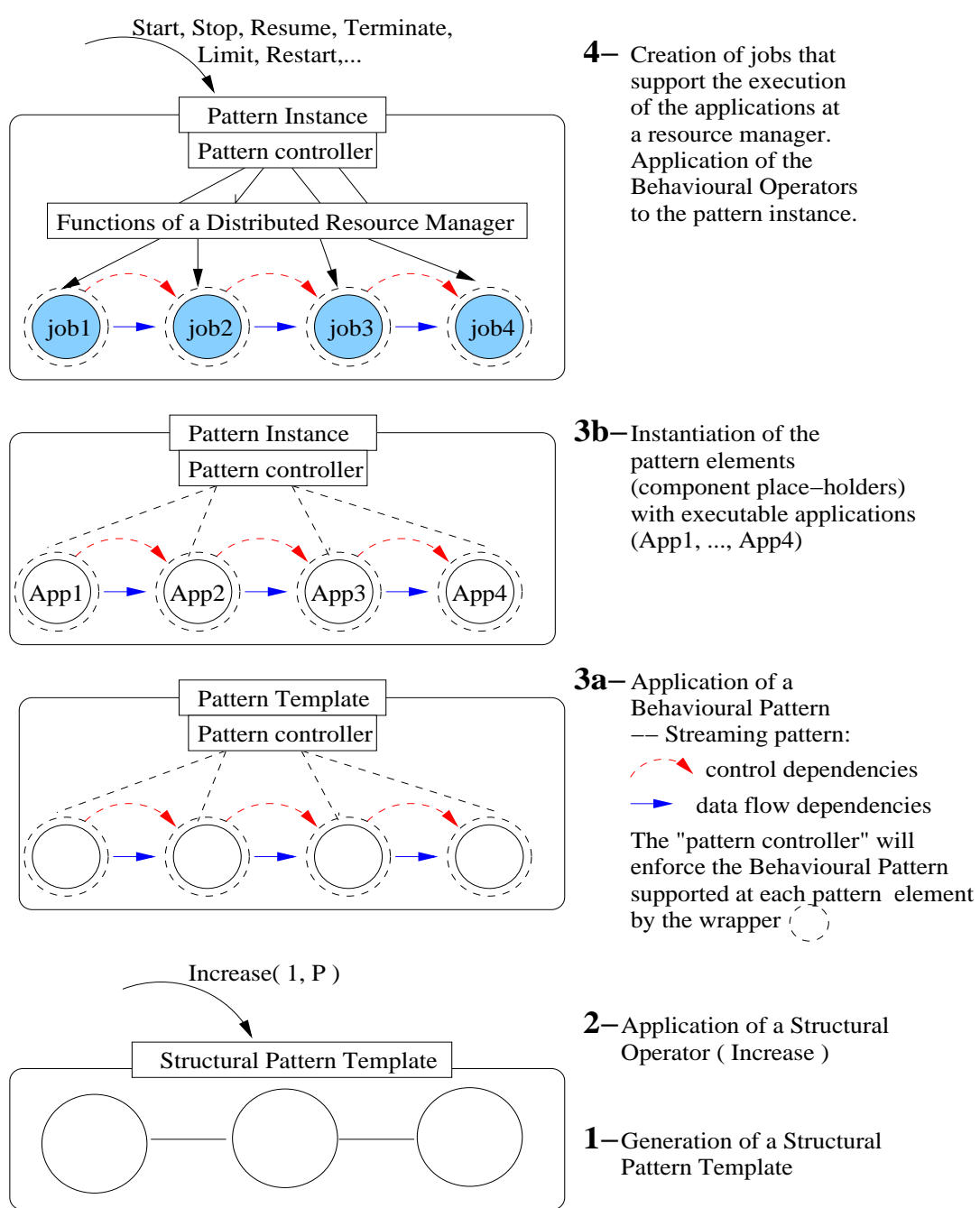


Figure 6.2: The necessary steps to configure and execute an application using patterns and pattern operators. Please read the Figure starting from the bottom.

Figure 6.2 presents those necessary steps to configure and execute a particular application which, in this example, is based on the *Pipeline* Structural Pattern combined with the *Streaming* Behavioural Pattern. Such configuration is typical of the processing/filtering in stages, the data produced by a scientific tool occupying the first stage of the pipeline. The Figure simply aims to be a logical outline of the more important entities and their interactions. The description of the four steps is:

1. The user defines the structural composition that will relate the particular executables. Particularly, the user selects a *Pipeline* Structural Pattern from a repository and generates one Pipeline Structural Pattern Template (S-PT) named with three elements, i.e. component place-holders (CPHs).

2. An extra element is added to the S-PT through manipulation by the *Increase(n, P)* operator version.
3. The user selects the *Streaming* Behavioural Pattern to be applied to all the elements of the Structural Pattern Template, and instantiates the CPHs with executables. Specifically:
 - Action **3a** in Figure 6.2 highlights the data and control dependencies between the elements defined according to the applied Streaming Behavioural Pattern through the *DefineBehavPatt(P, B-P)* operator. Such dependencies are enforced by an entity, at the pattern level, namely the *pattern controller*. As for the associated necessary dependencies between the pattern elements these are assured by *wrappers* enclosing each pattern element. The wrappers are therefore coordinated by the *pattern controller*. The execution of both the pattern controller and the wrappers is to be supported by the run-time system of a distributed Problem Solving Environment (PSE). Specifically, the mapping to a Grid/Web Services domain relies on the existing interfaces to suitable middleware.
 - Action **3b** in Figure 6.2, in turn, represents the *Pattern Instance* that resulted from the instantiation of the component place-holders to the selected executables (called *Applications* in the Figure). In a workflow-based PSE each *Application* may either represent a single executable or a group of executables organised in a taskflow.
4. Finally, the necessary Behavioural Operators are applied to the *Pattern Instance*. Successful execution of those operators is dependent on an adequate underlying distributed resource manager which is also responsible for the execution of the *Applications* represented as *jobs* in Figure 6.2.

Besides necessary capabilities such as selection of the most suitable hardware/software resources and data transfer across different authority domains, the distributed resource manager has desirably to provide checkpointing facilities supporting the suspension and resumption of the *jobs* (e.g. in [209] it is discussed an abstraction based Grid middleware layer supporting *checkpointable hierarchical Task Graphs*). Upon selection of the *Start* Behavioural Operator by the user, the pattern controller is responsible for using the available PSE API to launch all the *jobs* representing the *Applications* which are to be controlled by the underlying resource manager. Likewise, upon the invocation of the *Stop* Behavioural Operator, for example, the pattern controller is dependent on a suspend method available in the PSE's API to be applied to all jobs in the Pattern Instance.

6.2.2 Application Reconfiguration

In order to perform a few simple development and run-time reconfiguration actions upon the example in the previous sub-section, this sub-section describes the application of some of the concepts previously discussed in 5.2.1 and 5.3.

Reconfiguration at Development Time

After the fourth step in Figure 6.2, the user may decide to increase the number of stages of the pipeline with an extra *Application* placed as its last stage. The reconfiguration is done at

development time, meaning that either all components in the pipeline have already finished executing, or the user has explicitly requested their abortion through the *Terminate(pipelinePI)* operator.

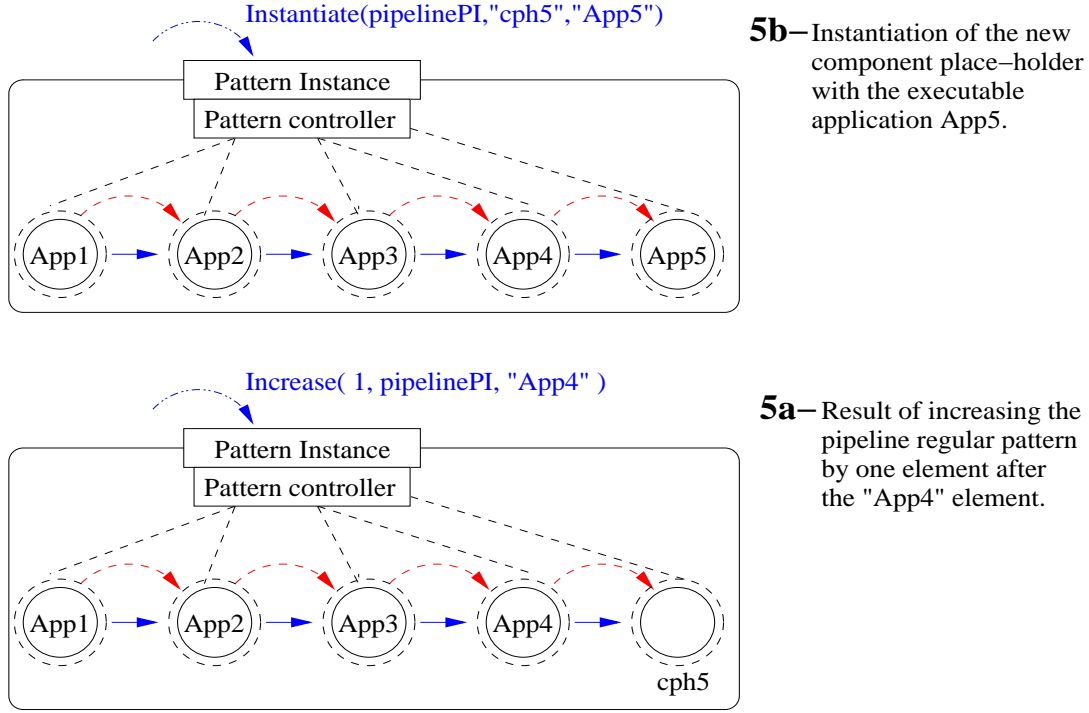


Figure 6.3: Application of the *Increase Structural Operator* at development time and after step 4 in Figure 6.2, in order to instantiate application App5 as the last stage of the pipeline.

Figure 6.3 presents the result of the cited reconfiguration, namely:

1. In step **5a** (at the bottom of the Figure), the user applies the *Increase(1, pipelinePI, "App4")* operator. The result is the generation of a new component place-holder (CPH), i.e. "cph5", specifically located after the last pre-existent component, i.e. "App4". Moreover, the new CPH is automatically annotated with the same data and control flows as the pre-existent components. Such is possible considering that the combination of the *Pipeline Structural Pattern* and the *Streaming Behavioural Pattern* defines a *Regular Pattern*, as discussed in section 5.2.2.

The addition of an extra stage to the "pipelinePI" in the characterised terms implies

- i) the creation of an extra *wrapper* to enforce the necessary dependencies to the previous stage; and
 - ii) the *pattern controller* becomes aware of that extra wrapper enclosing the new CPH so that it is also considered for the overall coordination of the pattern "pipelinePI", and according to the *Streaming Behavioural Pattern*.
2. In step **5b** (on top of the Figure 6.3), the new CPH, i.e. "cph5", is bound to the "App5" executable through the *Instantiate(pipelinePI, "cph5", "App5")*. Specifically, the last pipeline stage is annotated with the necessary information so that the "App5" component is executed as soon as an Execution Operator, e.g. *Start*, *Repeat*, etc., is again called upon the pattern "pipelinePI".

Reconfiguration at Execution Time

This section presents an example of modifying the application defined in Figure 6.3 while it is already executing. Specifically, we suppose that the user wants to modify the control flow between the executables in the pipeline to the *pull model* [9].

We recall that the “pipelinePI” may represent, for example, an application where the data produced by a scientific instrument, and processed by intermediate filters, is subsequently analysed in the last stage by a controllable visualisation tool manipulated by the user. The intended modification to the *pull model* may be particularly useful in the context of this example, in case the user wants to have direct control at the pace that data is transformed and analysed. Specifically, instead of data being continuously fed into the last stage, an on-demand request for data by the user, i.e. through that controllable visualisation tool named as “App5”, activates the penultimate stage, namely “App4” in Figure 6.3. This activation will still result on data being sent from “App4” to “App5” since the data flow is not changed. In case of insufficient data, subsequent activations will be propagated towards the previous stages (i.e. “App3”, “App2”), reaching, at last, the scientific instrument, i.e. “App1”, in the first stage.

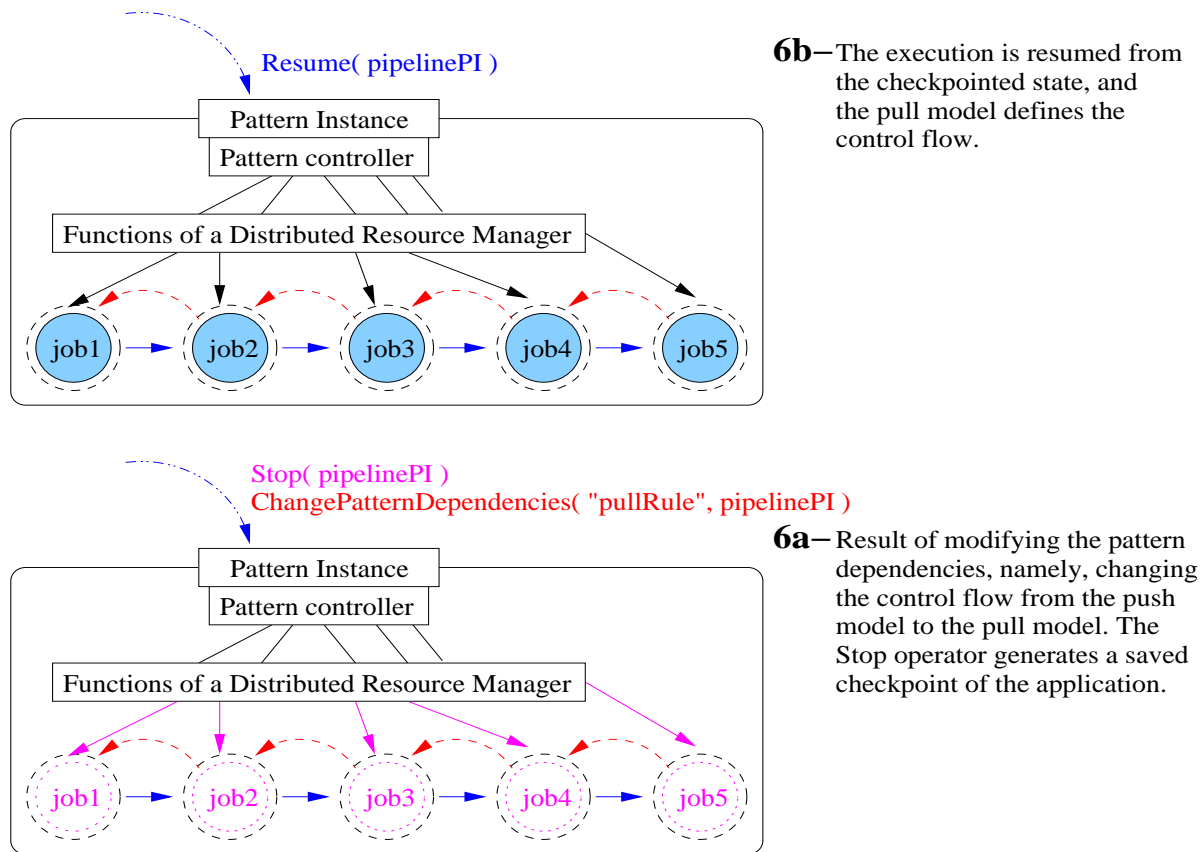


Figure 6.4: Modifying the control dependencies within the pattern, after the application in Figure 6.3 is executing.

Figure 6.4 represents the steps to change the control flow of the “pipelinePI” while it is already executing. Specifically,

1. Step **6a** (bottom of the Figure) presents the result of applying the following operator sequence:

- i) The *Stop(pipelinePI)* (Execution) Behavioural Operator generates the execution suspension of the running jobs that represent the *Application* components at run-time. To perform such operation, the *pattern controller* generates a call to a *suspend* operation which has to be available at the *Functions of a Distributed Resource Manager* layer presented in Figure 6.4. It is also assumed that the *suspend operation* causes the (coordinated) checkpoint of the state of all *jobs* in the Pattern Instance.
 - ii) To modify only the control dependencies in the Pattern Instance, the user applies the *ChangePatternDependencies("pullRule", pipelinePI)* Global Coordination Operator defined in section 3.3.6. The parameter "pullRule" in the operator represents the necessary set of coordination rules supported by the implementation system to modify the control flow.
2. Step **6b** (on top of the Figure 6.4) presents the result of applying the *Resume(pipelinePI)* operator to the application, which includes restoring the state of each of the "jobs" from the saved checkpoint and continue the Pattern Instance's execution.

In the following we will discuss the instantiation of the aforementioned generic architecture representing the inclusion of Patterns and Operators into a Grid-aware platform.

6.3 An Instance of the Architecture: Implementation over Triana

The present section describes how Patterns and Operators were supported by the Triana environment [20] towards a flexible configuration on application configuration and an explicit control over its execution and dynamic reconfiguration. The initial sub-sections describe the specific architecture for implementing Patterns and Operators in Triana, as well as the Triana environment itself, whereas the succeeding sub-sections describe the actual implementation of a sub-set of the proposed Patterns and Operators.

6.3.1 The Specific Architecture

The implementation architecture of the Patterns/Operators model outlines the concern of providing the user with high-level abstractions for application (re)configuration and execution control, hiding the difficulties inherent to (large-scale and heterogeneous) distributed execution. Certainly, those high-level abstractions

- a) demand an underlying system providing adequate support for the above requirements, specifically in Grid environments. For example, although the semantics of the execution operators is independent from a particular resource manager, their availability at run time relies on the possible capabilities provided by the selected resource manager made accessible by the underlying system;
- b) benefit from the availability of a suitable GUI for Pattern manipulation through the entire application development life-cycle.

The Triana system [20] provides support for the two concerns above and therefore is an adequate environment for the implementation of most concepts in our model.

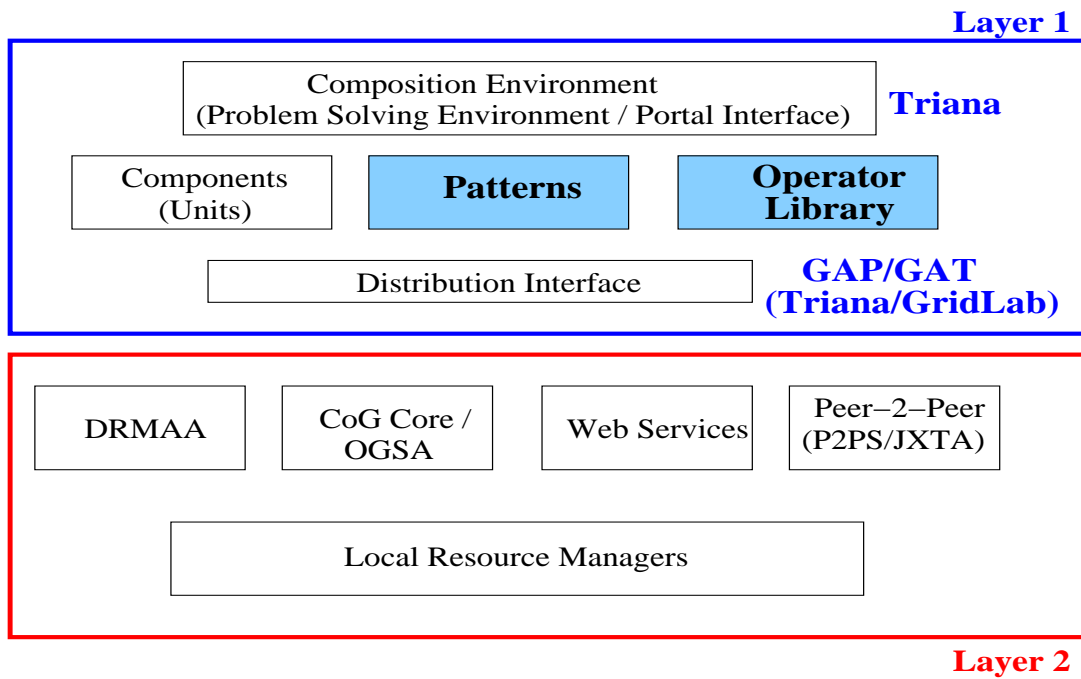


Figure 6.5: The specific architecture, based on the Triana environment, which supports the patterns/operators model. The shaded elements in the upper layer are the result of the work presented in this dissertation.

Triana is a Java-based workflow environment that supports application construction based on distributed components. On one hand, Triana provides the capability of decoupling the user interface from the distribution functionalities, and on the other hand, provides simple interfaces for different types of distributed execution and resource management, including the Grid. Namely, the Triana environment has been developed in order to provide the following kinds of execution:

- local versus remote;
- sequential versus parallel;
- Peer-to-peer execution;
- Web services;
- Grid computations.

Such characteristics are presented in Figure 6.5 which defines the specific architecture of the implementation of Patterns and Operators in Triana which benefit from Triana's pluggable architecture.

As depicted on the bigger box named *Layer 1* on top of the Figure, Triana provides a *Composition Environment* through its Graphical User Interface (GUI), from where the user may select the adequate components/services, which are identified as *Units*, and interconnect them in a workflow.

The Triana GUI was extended in our work so that the user may combine the components according to Structural Patterns defined in the Pattern repository, and manipulate them using the Structural Operators from the *Operator Library* (in the Figure). New patterns defined by the user can also be saved for latter reuse.

Concerning behaviour, Triana is inherently flow based, providing both a *Data-flow model* as well as *Control flow* mechanisms:

Data flow Data is sent in a stream or in frames between components in the workflow, and when all necessary data is available at a Triana Unit/component its execution is automatically triggered.

Control flow Control commands in Triana may control execution of the entire workflow as well of individual components (e.g. to trigger the execution of a component).

Therefore, the *Data-flow* in Triana matches the *Streaming* Behavioural Pattern, and also the *Producer/Consumer* although the buffer capacity between units is limited. Moreover, and due to the *Control flow* mechanisms, other coordination patterns are also possible in Triana, e.g., to simulate the *Client/Server* or the *Master/Slave* Behavioural Patterns.

The result of Structural Pattern refinement through Structural Operators and the subsequent composition with Behavioural Patterns is represented as a Triana workflow.

A set of Behavioural Operators is also available at the *Operator Library* in the specific architecture to control the execution of the final application configurations. Operators rely on Triana's commands/control mechanisms and capabilities of the *Distribution Interface*, which is represented in the specific architecture in Figure 6.5. Concretely, the *Distributed Interface* supports workflow enactment and Triana commands over different types of distributed execution environments without the burden of dealing with their specificities. Triana provides a few implemented bindings for distinct distributed execution environments, but other bindings may also be developed and plugged into the architecture.

In Triana, the GUI is completely decoupled from the distributed execution support:

- The GUI is used to define units, to produce taskgraphs, and to submit commands. Their concrete execution is in turn processed by the *Distribution Interface*, which is represented in Figure 6.5.
- There are different available readers/writers for units, taskgraphs, and commands, and new readers/writers may also be seamless inserted into the architecture. For example, task graph writers include BPEL4WS [267] and a proprietary XML format.

The decoupling of the GUI from the distributed execution provides, for example, the usage of the Triana GUI simply to produce a taskgraph in a specific language, or Triana is only used to execute an externally defined BPEL4WS taskgraph.

The Triana's *Distribution Interface* is supported by the *GAP/GAT* interface [40] (see Figure 6.5), developed within the *GridLab* project [41,42]. The *Grid Application Toolkit* (GAT) [39,41] provides an abstraction layer to construct and execute Grid applications which are independent of the underlying middleware actually deployed. Previous knowledge of the runtime environment is therefore not mandatory for end-users and application developers. The GAT

API provides abstract capabilities commonly required by Grid applications like resource discovery, job submission, or file transfer.

In turn, the *Grid Application Prototype Interface (GAP Interface)* [40], which was considered for the implementation of our model, is compliant to the GAT interface but also enables cross-environment support for both Grid, Web Services and Peer-to-Peer technologies (see Figure 6.5). Namely, the GAP interface provides applications with methods for advertising, locating, and communicating with other peers/services.

Specifically, in the Triana version used for the implementation of Patterns and Operators (Triana version 3.1), there are three middleware bindings implemented for the GAP Interface [40,83,84] (see Figure 6.5):

- JXTA [77], a peer-to-peer toolkit originally developed by Sun Microsystems;
- P2PS, a lightweight alternative to JXTA;
- and a Web services binding.

A XML-based interface presented by the components in the Triana toolbox may be dynamically bound to one of the mentioned supported underlying middleware. The task graph generated by the composition tool is also defined in a proprietary XML, and subsequently bound to one of the described middleware.

The first developed GAP interface was to JXTA which provides protocols for peer-to-peer discovery and communication. However, this first binding proved to have some performance and reliability problems. Therefore, a second binding was developed, namely the P2PS middleware [202], whose architecture was inspired by that of JXTA. The P2PS binding [266] provides lightweight but effective Peer-to-Peer mechanisms which are based on XML [76] advertisements and messaging. For this reason, the P2PS infrastructure is independent of any implementation language and computing hardware, and it is also not tied to any single transport protocol. With adequate P2PS implementations, Triana is to support the building of a P2PS network that includes everything from super-computer peers to PDA peers.

The third GAP binding was the Web Services binding [83] which is based on UDDI registry [85] and the *Web Service Invocation Framework (WSIF)* [86]. Triana applications may therefore discover, publish, and invoke Web Services, where data is packaged from Triana units to Web Services. In the Triana GUI, Web services are represented as *Units* which provide transparent invocation, and Triana workflows define service composition.

Concerning Grid access, the flexibility of the GAP interface allows applications to work with Grid middleware such as *Open Grid Services Architecture (OGSA)* [48,60], the *Java Commodity Grid (CoG)* kit [13], and the *GridLab Resource Management System (GRMS)* service [80,81,84]. Implementation-wise, the Triana mappings to OGSA (through the GAT interface) and to CoG were still not included in version 3.1 although they were already under study. Already under advanced development was the binding to the GRMS service [84] as part of the GridLab project.

The GRMS service [81] is an open source meta-scheduling system to support management to the whole process of remote job submission to various batch queuing systems (e.g. Condor [268], PBS, Sun Grid Engine), clusters, or resources directly. For example, GRMS provides

dynamic resource selection and mapping for load-balancing among clusters. The first release of GRMS implementation is based on Globus system.

Finally, a mapping was also under study of the GAP interface to the DRMAA API [43] distributed resource manager system. Nevertheless, section 6.5 describes, in particular, a possible mapping of the Behavioural Operators to the DRMAA API. This aims to clarify how specific Operators like *Stop* and *Resume* can be effectively supported as long as there is an API with adequate operations for distributed job control.

6.3.2 The Triana Environment

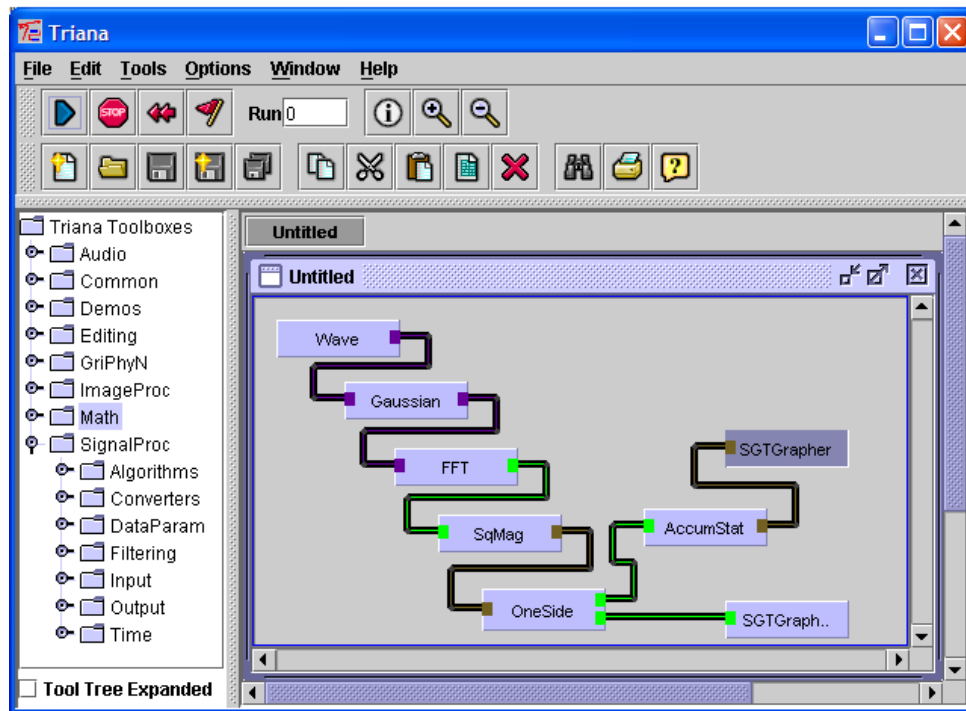


Figure 6.6: *The Triana's Graphical User Interface.*

In general, Triana [20, 40, 84, 203] is a Java workflow based Problem Solving Environment written for application construction based on distributed components. Through the Graphical User Interface provided with Triana (Figure 6.6), users have access to components representing services/tools for many different areas and that can be easily composed for building scientific applications. For example, there are components for signal processing, image manipulation, mathematical calculations, etc., and they are later bound to the tools/services that they represent to create a highly dynamic programming environment. New components may be also easily added to Triana (e.g. using a component wizard).

On programming a component to be added to Triana, for example a scientific tool, the user specifies the type of information a component can receive as well as the type of information it can output. Specifically, the interface to a component in Triana is well defined through *input and output ports* (designated as *nodes*) and *parameters*. Typically, input/output nodes allow the connection to other components through communication channels, whereas parameters allow

modifying and sending settings or information not directly related to generated data results among components. Each component has to be compiled only once and thereafter component communication obeys its specified interface.

Components are present in the toolbox on the left-side of the Triana GUI (Figure 6.6) and they are designated as *Units*. A Triana Unit is defined in a XML proprietary language to represent the necessary interface to the associated tool/service as well as all the necessary information for their late binding.

Users drag and drop *Units* from the toolbox onto the scratch pad (present on the right side on the toolbox), and create workflows by dragging cables that connect components together. Sender components are connected through output ports (or *nodes*) on the right-side, to receivers' input ports (*nodes* on the left-side). Users may also group selected components together into a component which represents the entire set. This "group component" also has input/output ports and parameters for connecting the group (and some of its hidden elements) with other components in a workflow.

In the Triana GUI, the *user interface* to access the parameters of a component is designated *parameter panel*. In Triana, this user interface is decoupled from the component and may be run in a different computer. The parameter panel allows tuning the value of interface component variables (e.g. to change the wave length/frequency of a wave generator tool). In the case of group components the parameter panel also gives access to all parameters of each of the individual components belonging to the group.

Data and Control Flow

Data and control flow between components/Units in Triana may be defined through Unit's ports/nodes, events, and control commands and Units.

The available component nodes in Triana are:

Data nodes These define the data flow connection between two tools/services, where data is sent along the output nodes of the sender, to the input nodes of the receiver. Data nodes have associated data types, and Triana provides design-time type checking since only allows the connection between between output and input nodes in case of data type matching.

Data flow along data nodes is also related to control flow since data arrival at a component's input node may trigger its execution. Specifically, input data nodes may be defined as *mandatory/essential* or *optional*. Mandatory input nodes block the execution of a component until data is received on that node. Optional input nodes, on the other hand, allow component execution triggering (e.g. through control operations) even in the absence of data. This means that if the component has several mandatory data input nodes, only when data is received on all of them is the execution of the component triggered. Contrarily, if a component has several optional input nodes its execution triggering is independent of the arrival of data to those nodes. Data arriving at an optional data input node of a running component may be either consumed by the component or simply ignored.

Trigger nodes These support control flow between components where there is no specific data dependency. Control flow between two components is enabled by defining a *trigger input node* at one of the components to which the other component sends a control signal. Specifically, any input sent by the (controller) component will trigger the execution of the (receiver) component containing the trigger node (the value of the input is ignored). Input trigger nodes may also be defined as optional or essential (mandatory). In Triana version 3.1 an input trigger node is only effective if it is defined as mandatory. This means that a control signal sent to an optional trigger node has no effect. On the other hand, an effective trigger node (i.e. mandatory) blocks the execution of a Unit, i.e. even if a Unit has data at all its *essential data nodes*, in the presence of an essential trigger node, the Unit's execution will only be triggered by explicitly sending a control signal to that essential trigger node.

Parameter nodes Besides available from the *parameter panel* the user interface variables are also accessible through *parameter nodes* allowing sending their values between components. Parameter nodes may also be classified as input and/or output. A *parameter input node* allows changing the value of the variable it is associated to. A *parameter output node* is used to send a variable's value to another component. Parameters nodes may be connected through a cable to both data nodes and parameter nodes in another component. In this case, it is possible to synchronise the values of two variables (i.e. parameters) in two distinct components by connecting their output and input parameter nodes.

Input parameter nodes may optionally also be defined as *triggering* nodes. Such means that as soon data is ready at a component's input parameter node, its execution is triggered.

The connection between nodes is supported by communication "pipes" through which Data and Control messages may be transmitted from the sender to the receiver, e.g. causing the receiver to execute.

Data and control flow in Triana is also possible through events. For example, each Unit's parameter, when modified, generates an event. Other Units may be defined as *listeners* to those parameter events. On parameter update, all the listeners are notified and receive the new value for the parameter. Triana supports this notification also to distributed Units/components in the network. The fact that a Triana Unit is a listener to its own parameters allows the referenced decoupling of the parameter panel from the Unit itself for remote tuning.

Additionally, Triana provides looping and logic Units (e.g. "if-then-else" Units and "do-while" Units) to graphically control the dataflow.

Finally, the Triana GUI provides control commands to execute the workflow defined in the canvas, e.g. to start or to abort the execution of the workflow.

Local and Distributed Execution Models

Conceptually, and as previously discussed, Triana is a two-layered application where the Triana GUI is de-coupled from the provided distribution functionalities. These are supported by different types of distributed execution environments, including Grid environments. A detailed description of Triana's architecture is presented in [21, 72, 82, 84, 269]. In the following

we present an introductory description of Triana’s execution model, particularly peer-to-peer distributed execution in Triana (useful in highly dynamic environments).

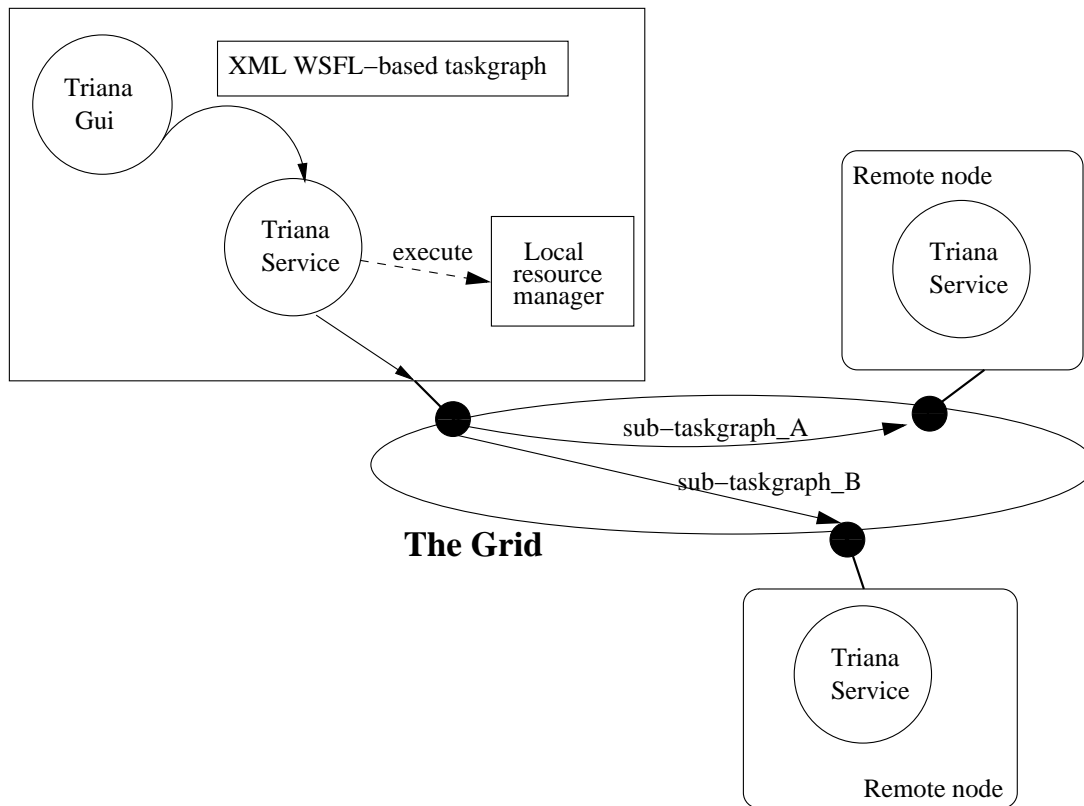


Figure 6.7: A simplified vision of Triana’s distribution model.

Figure 6.7 presents, in a simplified way, how Triana provides large-scale high-performance execution support to applications. Specifically, distributed execution results from the composition of several network peers collaborating to solve a problem [21,72]. Each peer acts both as a client (for local users’ requests) and as a server (for remote peers’ execution requests). Access and communication between peers is based on JXTA architectures [77] (i.e. for distributing and locating available peers through resource discovery, and including pipe based communication, rendezvous nodes, etc). Locally, each peer may access existing Grid services, for example, to execute high-performance computations (e.g. through the access to the GRMS service).

The Triana implementation considers two types of components: the *Triana Controller (TC)*, and the *Triana Service (TS)*. The *Triana Controller* represents the user interface (either based on command line or on a GUI) and provides access to a network of Triana service daemons running on multiple CPUs. The *Triana Service* is responsible for the distributed execution by accessing other peer *Triana Services*. Typically, one *Triana Service* is in direct communication with the *Triana Controller* and it can be either local or remote. In Figure 6.7, such *Triana Service* is local to the *Triana Controller*, i.e. Triana GUI.

Through the GUI, users define an application by connecting a set of components forming a workflow. Users may also define which parts of the workflow should be executed on remote peers. Consequently, the peer supporting the Triana GUI acts as a co-ordinator for launching different parts of the application on other peers.

The interaction with the GUI results in a *TaskGraph*, i.e. a “work-flow process definition

that defines the tasks that need to be executed and the order in which they are executed” [73]. Namely, each component in Triana is the unit of execution resulting in a *Triana Task*, and the *TaskGraph* is an internal object based workflow graph representation defining the Tasks (also representing compound components) and their interconnections. The taskgraph is written in a proprietary XML-based language, but Triana also provides a taskgraph writer for BPEL4WS, and other writers can be plugged into the architecture as well.

The resulting *TaskGraph* is passed to the *Triana Service* directly in communication with the *Triana Controller* which decides which parts are to be executed locally and which *sub-TaskGraphs* should be sent to remote peers. Each peer must support a *Triana Service*, to enable execution requests to be received from the co-ordinator – essentially the Triana Service acts as a hosting environment to launch and manage task execution on remote resources. A peer may subsequently also sub-contract execution to other peers.

Each *Triana Service* itself consists of three components: a *Command Server Process*, a *Client* and a *Server*. Typically, the first two components are only active in the *Triana Service* directly in communication with the *Triana Controller*. Specifically, the *Command Server Process* interacts with the *Triana Controller*, and the *Client* is responsible for sending requests to the remote peers in case the required programs/services are not available locally, and for collecting their replies. Otherwise, the *Server* component executes/accesses those local programs/services itself or contacts a local resource manager (e.g. the Globus GRAM [14]¹ or GRMS [80]). The *Triana Services* at the contacted remote peers only work in server mode and therefore, upon receiving a *sub-TaskGraph* (as represented in Figure 6.7), their *Server* component evaluates if the necessary programs (services) are available. If not, the *Server* contacts the *Server* of another peer for remote execution. Each peer *Triana Service* may therefore act as a gateway for distributed execution.

Additionally, Triana provides the user with the possibility of specifying custom distributed policies defining the mechanism for distribution within a group of Units (i.e. Tasks). Namely, the unit of distribution is the Group Unit (e.g. representing a *sub-TaskGraph*), and it has its own distribution policy implemented as a Triana Unit (named *Control Unit*). Triana provides some distributed policies by default, e.g. parallel (task farming with no communication between resources) and pipeline (each Unit in the Group is distributed on a different resource and data is passed between them).

The following section describe the extension of the Triana environment to include support for our Patterns and Operators.

6.4 Patterns and Operators in Triana

The Triana PSE tool was augmented in our work with a *Pattern Template (PT)* and *Operator* library as represented in Figure 6.8. The user may therefore select a Structural PT from the library, and may apply one or a combination of operators to modify the structure of the template. As previously described, the *Structural Operators* provide a transformation between patterns,

¹GRAM provides a single protocol for communicating with different batch/cluster job schedulers, and enable users to locate, submit, monitor and cancel remote jobs on Grid-based compute resources

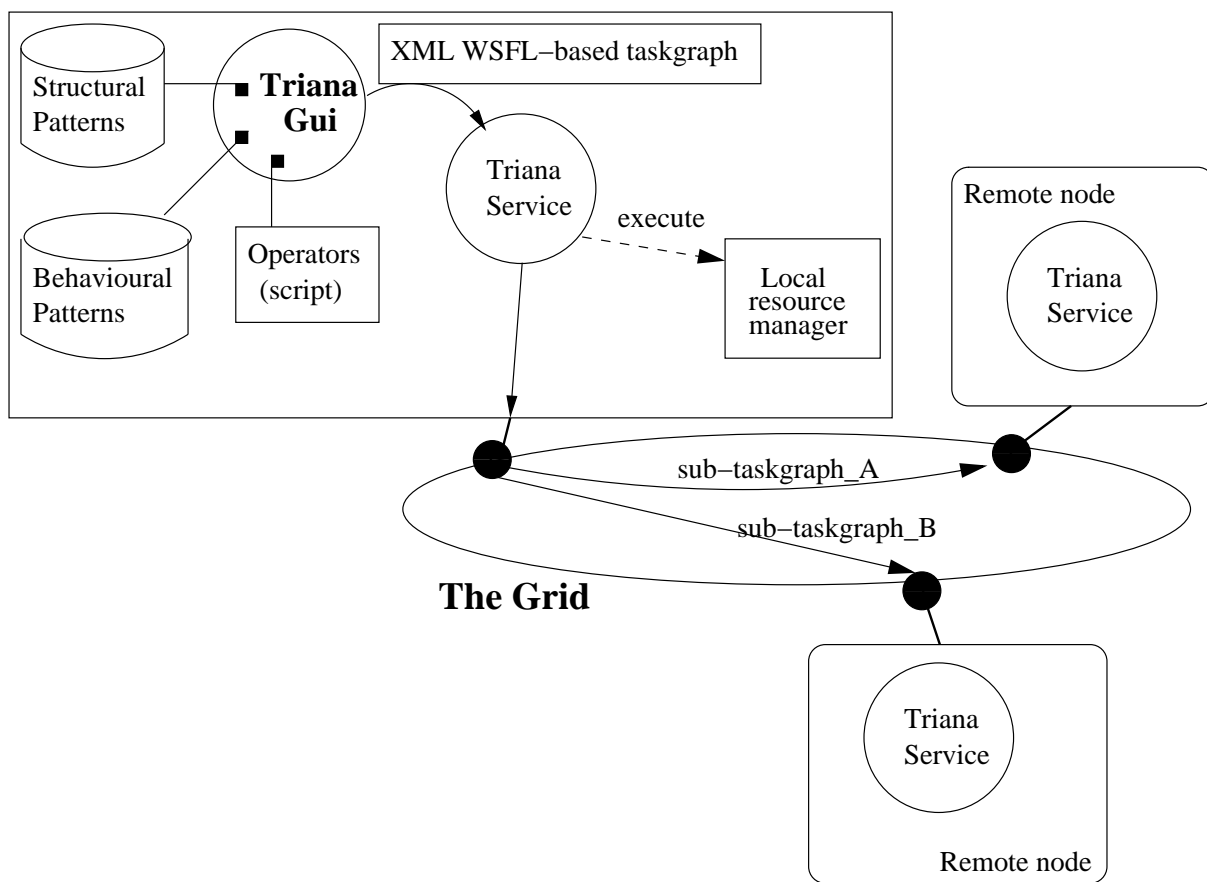


Figure 6.8: *The inclusion of Patterns and Operators into the Triana Environment.*

and are invariant to a given PT structure. The result may be stored by the user as a new template in a user-defined PT library.

Once the structure has been defined, the user now instantiates components accessible through Triana to the elements of a PT. This is then followed by defining interactions between components – based on the provided *Behavioural Pattern Templates*, generating a *Pattern Instance*. The current available Behavioural Patterns are the *Producer/Consumer* and *Streaming* which are provided by Triana by default. Due to Triana’s controlling mechanisms for independent control flow from data flow, other coordination patterns may also be defined (e.g. simulating the *Client/Server* pattern). Subsequently, the component interactions may be modified using the Behavioural Operators.

Additionally, and in order to simplify and automate applications construction, the implementation supports the usage of simple scripts as well. The scripts allow the creation of Structural Patterns and their manipulation through Structural operators and execution control through Behavioural Operators.

To conclude, the Triana’s distributed execution model, as represented in Figure 6.8 and previously described, supports the defined pattern-based component interactions and execution control in a distributed network of peers which may give access to Grid services.

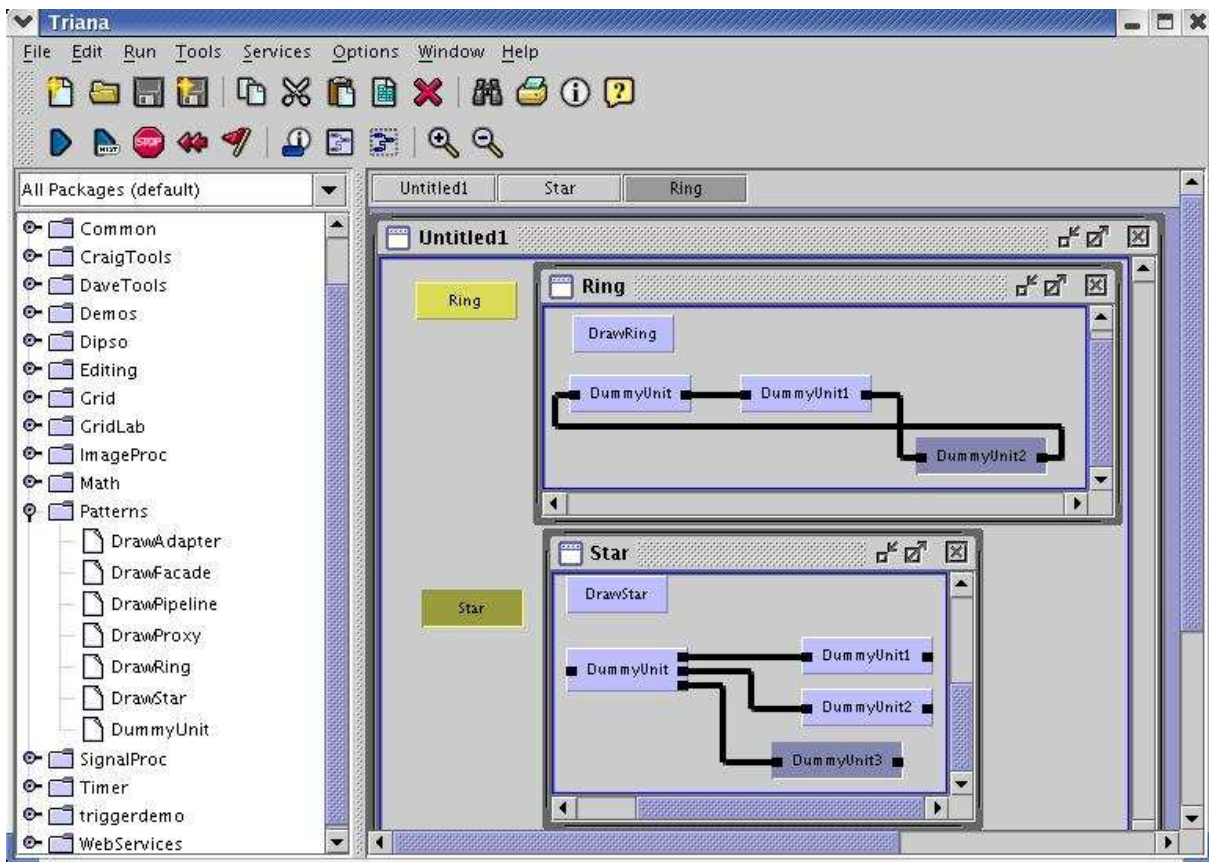


Figure 6.9: *The Triana's Graphical User Interface.*

6.4.1 Structural Patterns and Operators in the Triana GUI

Structural Patterns were made available in Triana's toolbox as normal components (from a graphical perspective) as presented in Figure 6.9.

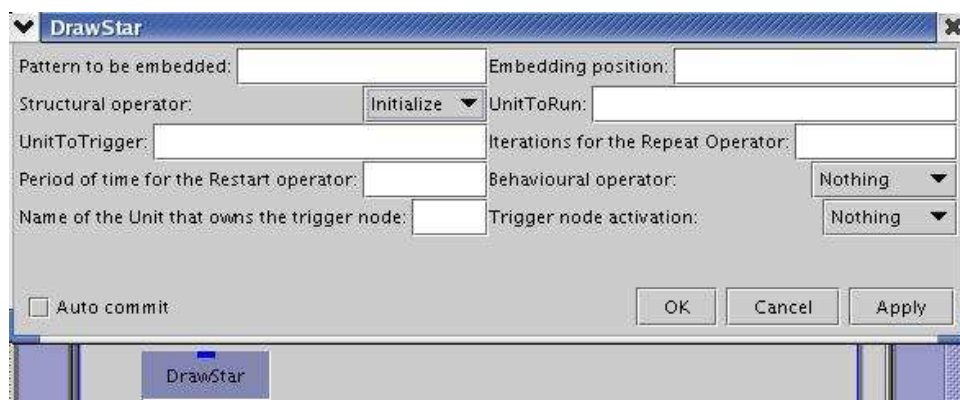


Figure 6.10: *Initialisation of Pattern Template.*

The user just has to drag and drop them into the scratch pad and subsequently initialise them, as represented in Figure 6.10. Figure 6.9 shows a Ring Pattern Template (PT) and a Star PT that resulted from the initialisation of *DrawRing* and *DrawStar*, respectively. The user may change the name of each PT through the Triana GUI and in case of name collision the name is

automatically appended with a number defining a unique identifier.

Each Pattern Template represents a set of component place-holders called *DummyUnits* which can be instantiated to other PTs or tools from the toolbox. *DummyUnits* are connected together according to the PT's specific Structural Pattern (i.e. Ring, Star, etc). Each component place-holder, i.e. *DummyUnit*, has a unique identifier within the Pattern (e.g. *DummyUnit*, *DummyUnit1*, and *DummyUnit2*, as in the Ring in Figure 6.9).

Structural Operators are available as parameters to Pattern Templates (through a parameter panel) which upon selection act over the entire PT producing the required transformation. The structural constraints of the specific operator are obeyed even if the PT is an *Hierarchic PT* (i.e. this PT already comprises other PTs as its elements). For example, the *Increase* operator applied to a Pipeline PT adds one extra element to the PT, independently of the stages being *DummyUnits* or other PTs. In this case, the position where the new component place-holder is placed within the PT is pre-defined, i.e. our implementation in Triana does not yet support the Increase operator version *Increase(n, P, position)* (the semantics of this version was presented in section 3.3.2).

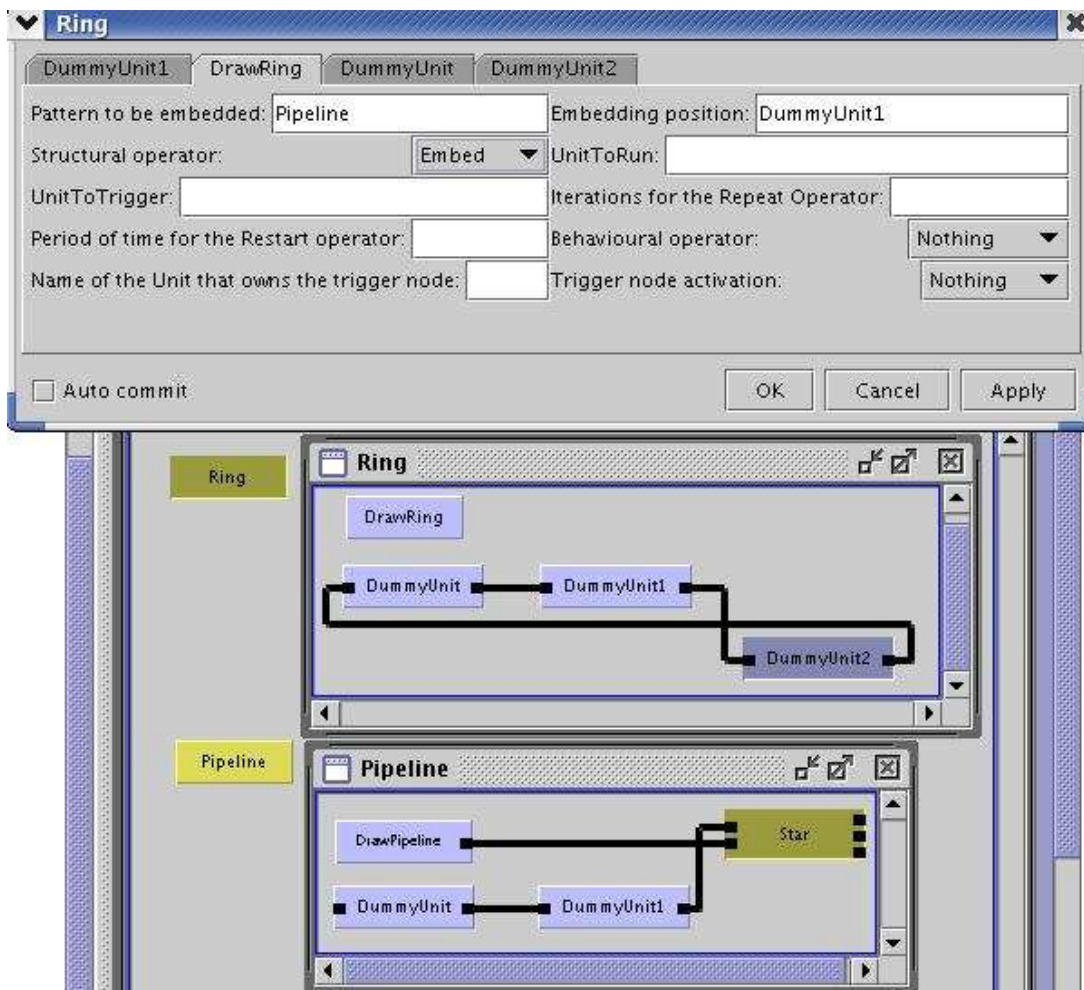


Figure 6.11: Application of the Embed Structural Pattern to the Ring Pattern Template.

An example of the application of a Structural Operator to a Structural Pattern is presented in Figure 6.11. Namely, to apply the *Embed* operator to the Ring PT, the user must first invoke the Ring's PT parameter window, as shown in the Figure. Next, the user specifies that the Pipeline

PT should be embedded into the Ring PT's component place-holder named "DummyUnit1". In this example, the Pipeline PT already has an embedded Star PT.

In order to implement Structural Pattern Templates in Triana, these were mapped to *groups* of the Triana model (i.e. a *Unit* enclosing other *Units*). Each group supporting a PT contains:

- a) the connected DummyUnits;
- b) the *pattern controller* which supports the Pattern's management.

For example, the Triana's group forming the Ring PT in Figure 6.12 includes one component place-holder (CPH) still uninstantiated named *DummyUnit*, two already instantiated CPHs named *Pipeline* and *MakeCurve*, and the pattern controller (named *DrawRing*).

Concerning structural issues, it is the responsibility of the *pattern controller*

- to keep track of the number of elements within the PT (and their connections);
- to listen to relevant events (e.g. requests to instantiate *DummyUnits* to tools);
- to support the execution of the Structural Operators.

Examples of these actions are described next.

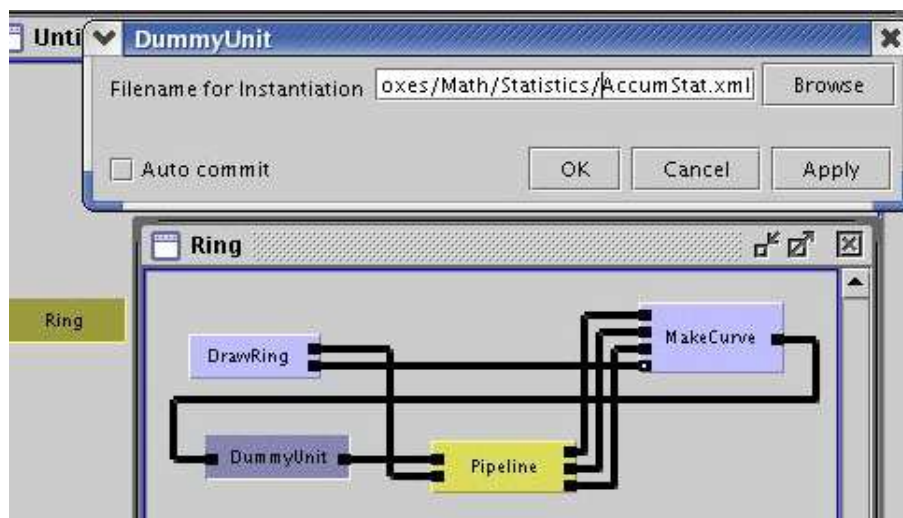


Figure 6.12: Instantiation of the *DummyUnit* component place-holder to the *AccumStat* Unit.

Namely, the instantiation of a particular component place-holder like the *DummyUnit* in the Ring pattern in Figure 6.12 requires the activation of its parameter panel in order to select the required Unit (representing a particular Tool, Service, Pattern, or Workflow) from the Toolbox.

Upon selection, an event is generated at the *DrawRing* pattern controller which replaces the *DummyUnit* with the selected Unit while guaranteeing the necessary connections and keeping the structural constraints. Figure 6.13 shows the result of that instantiation.

On the other hand, the action resulting from the selection of a Structural Operator at the *DrawRing*'s parameter panel is restricted to that operated Ring. In order to manipulate one embedded Pattern, for example the Pipeline pattern in Figure 6.13, the user has to: activate the parameter panel of the Pipeline pattern, specifically the *DrawPipeline*'s parameter panel;

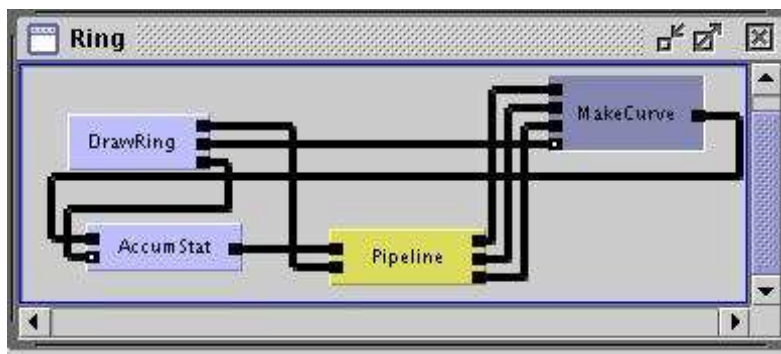


Figure 6.13: A Ring pattern fully instantiated at the outmost level.

and select the desired operator. The same applies to other patterns embedded into the Pipeline pattern. Additionally, the parameter panel of all embedded patterns are also directly accessible from the parameter panel of their enclosing pattern. In this way, the encapsulation of each Pattern into a Triana's group provides a layered access to all Patterns forming a Hierarchic Pattern.

After composing an application by combining PTs with existing components, the user can save them as a group component in the toolbox for later reuse.

6.4.2 Scripts of Structural Patterns and Operators

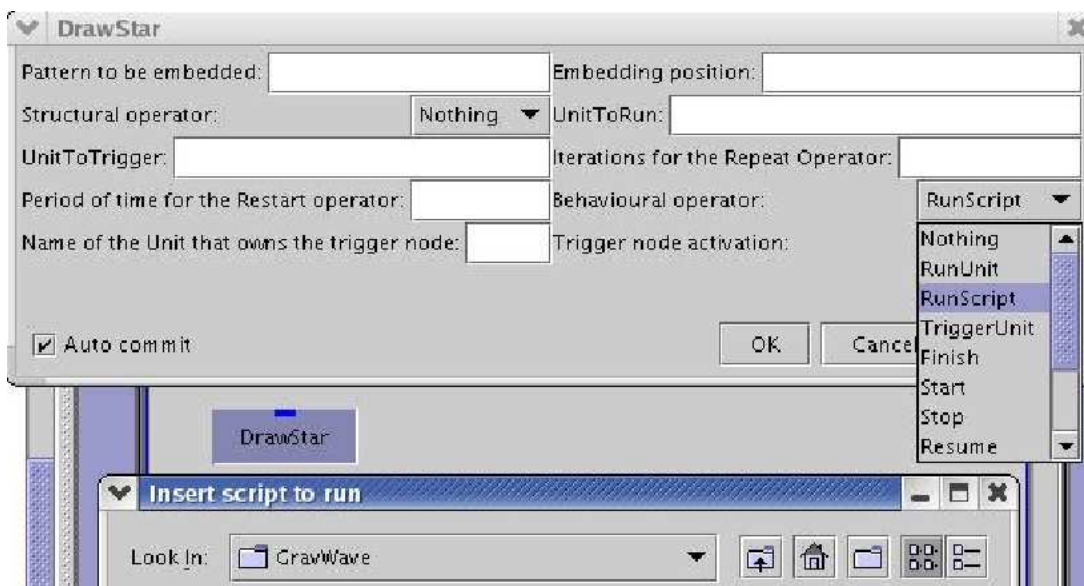


Figure 6.14: A script with structural operations is associated to a particular pattern.

In order to allow the automated building of an application's configuration the user may define the creation and manipulation of Structural Patterns in a script. The script is processed by selecting the *RunScript* operation in the parameter window of a pattern controller and by defining the script's name as presented in Figure 6.14. The structural operations defined in the script are processed in the context of a particular Structural Pattern. Additionally, the scripting

process has a recursive definition meaning that from within a script it is also possible to process (sub)scripts associated with other Structural Patterns. The following resumed description of the semantics of the scripting actions are illustrated in the *Extended Backus-Naur Form (EBNF)* meta-language [270].

A - Common Structural Operations

There are a few actions which are common in structural scripts, as defined in the EBNF graph in Figure 6.15:

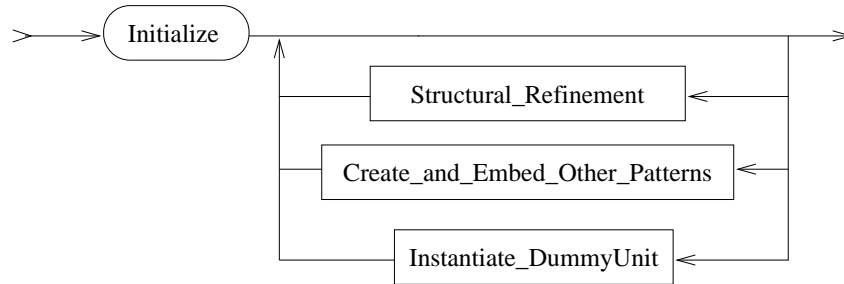


Figure 6.15: General structural manipulation from a script defined in EBNF. The presented actions (nonterminal EBNF elements) may be interleaved and applied as many times as necessary. A terminal element defining the end of script processing is omitted for simplification reasons.

1. Although a script is usually associated with an existing Pattern Template on the Triana GUI, it is also possible to generate the Pattern Template itself from within the script. Such is possible as long as the Triana Task supporting the pattern controller is already executing. This pattern generation supports the *Create(SP, name [, nElems])* Structural Operator presented in section 3.3.2 and it is defined as the *Initialize* action in the script. This pattern initialisation is defined as necessary in the EBNF graph in Figure 6.15 (i.e. the *Initialize* terminal element), although such is not mandatory.
2. In a script associated to a pattern, the user may apply to that (newly created) Pattern Template as many Structural Operators as desired. These are represented by a nonterminal element named *Structural Refinement* in the EBNF graph in Figure 6.15. The *Structural Refinement* is defined in Figure 6.16². The nonterminal element *Number_Max* in this graph represents the number of component place-holders in the *Increase/Decrease* operators.

For example, to create a Star with four component place-holders, the user as to:

- (a) drag and drop the *DrawStar* Unit into the canvas;
- (b) launch the processing of the following script from the parameter panel of *DrawStar*:

```
Initialize
Increase 1
```

²Although not defined here, the *Reshape* Structural Operator was also implemented for the non-topological Structural Patterns and it is accessible from the Triana GUI.

As for the *Embed* operator, the nonterminal element *Embed_Pattern* in the EBNF graph in Figure 6.16 represents the embedding of an already existing pattern into a component place-holder (i.e. *DummyUnit(i)*).

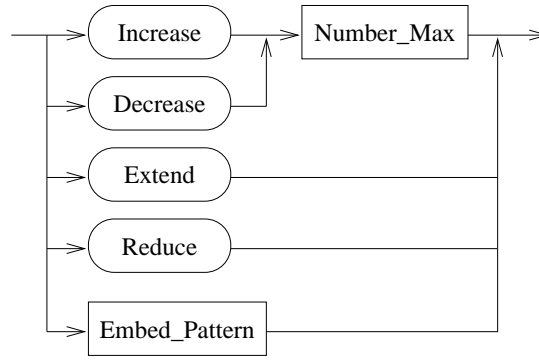


Figure 6.16: *Structural Operators*.

The definition of the *Embed* Operator is presented in Figure 6.17.

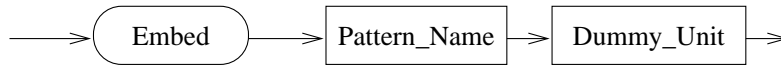


Figure 6.17: *EBNF definition of the usage of the Embed Structural Operator from a script*.

Concretely, in the *Embed* operation the user has to:

- a) specify the name of an already existing pattern (represented by the nonterminal element *Pattern_Name* in Figure 6.17);
- b) define where this pattern should be embedded, i.e. into which component place-holder. This one is represented in Figure 6.17 by the nonterminal *Dummy_Unit* EBNF element, which represents the names of the available *Dummy_Units*. Figure 6.18 defines that the

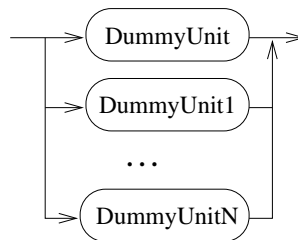


Figure 6.18: *EBNF definition of the names of the component place-holders within a Pattern Template*.

possible names of the component place-holders within a generic Pattern Template in Triana are *DummyUnit*, *DummyUnit1*, *DummyUnit2*, etc.

For example, the following script performs the embedding operation previously presented in Figure 6.11, if this script is processed in the context of the pattern controller *DrawRing* in the Figure:

3. As previously presented in Figure 6.15, another common action within a pattern and operator based script is the instantiation of the available component place-holders to tools in the Triana toolbox. Figure 6.19 presents the definition of the instantiation operation from a script.

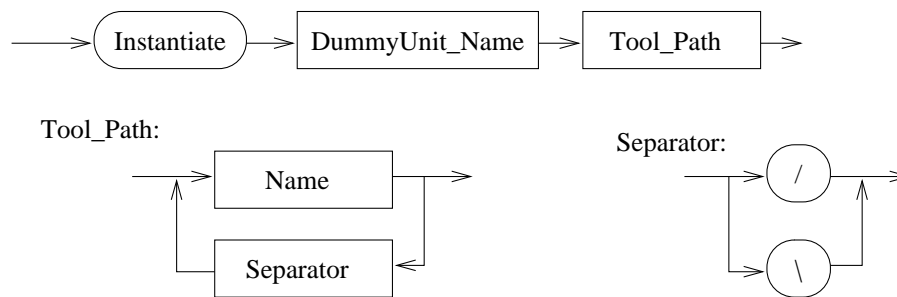


Figure 6.19: EBNF graph for the “*Instantiate_DummyUnit*” nonterminal element in the graph in Figure 6.15.

For example, the following script line performs the instantiation operation previously illustrated in Figure 6.12.

```
Instantiate DummyUnit
    /home/mcg/working/toolboxes/Math/Statistics/AccumStat.xml
```

4. Finally, the nonterminal element *Create_and_Embed_Other_Patterns* in Figure 6.15 represents the possibility of creating other Structural Pattern Templates, manipulate them with Structural Operators, and subsequently embed them in the pattern processing the script. Moreover, the user may operate the principal pattern before or in between (and after) those actions. This is defined in Figure 6.20.

On one hand, the *Create_Pattern* nonterminal EBNF element is illustrated in Figure 6.21. Please note that although the *Facade*, *Adapter*, and *Proxy* Design Patterns are presented in this Figure, and also illustrated in an example in section 7.7.1, they are not yet fully implemented in Triana.

On the other hand, the nonterminal EBNF element *Run_Structural_Script* in the graph in Figure 6.20 represents the processing of a *sub-script* defining the structural manipulation of the newly created PT. This is described next.

B - Processing a sub-script

In order to trigger the structural manipulation of a PT from a script being processed by the principal pattern, the user has to:

- a) define a *sub-script*, i.e. enclose those manipulations with the *RunStructuralScript* and *EndStructuralScript* script actions;

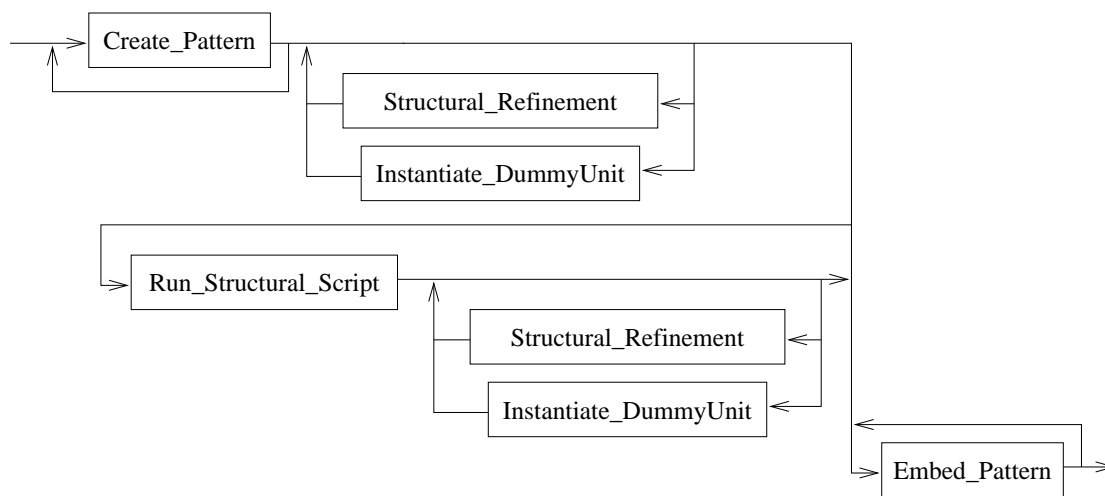


Figure 6.20: EBNF graph for the “Create_and_Embed_Other_Patterns” nonterminal element in the graph in Figure 6.15.

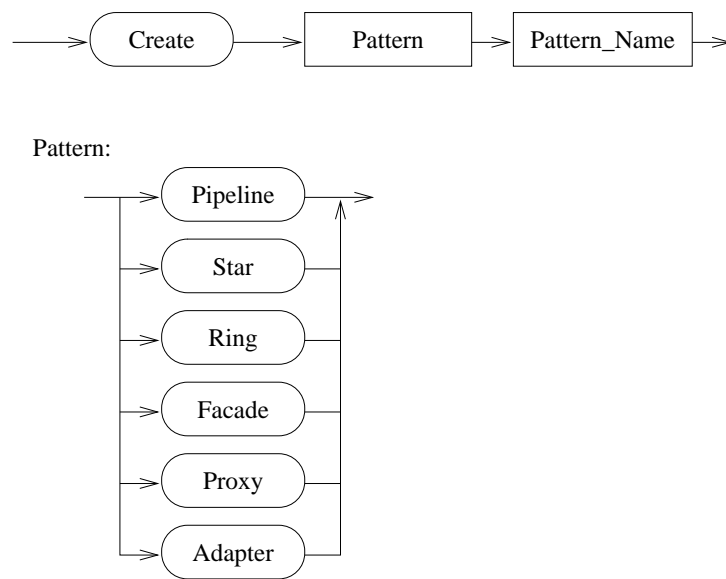


Figure 6.21: EBNF graph for the “Create_Pattern” nonterminal element in the graph in Figure 6.20.

- b) identify the PT to which that sub-script is applied. The sub-script will be processed independently by the pattern controller of that PT.

For example, the following script illustrates the definition of a sub-script, which is preceded by the creation of the PT it is applied to; subsequently, this new PT is embedded in one component place-holder:

```

Create Star Star
RunStructuralScript Star
  Increase 1
EndStructuralScript
Embed Star DummyUnit2
  
```


Assuming that this script is processed by a Pipeline PT with three component place-holders, the result is (a similar Pipeline PT was illustrated in Figure 6.11):

- a) the creation of a Star PT named “Star” to which a new component place holder was added through the *Increase* operator;
- b) the resulting “Star” PT is subsequently embedded in the “DummyUnit2” component place-holder of the Pipeline PT.

Figure 6.22 defines the graph for a sub-script.

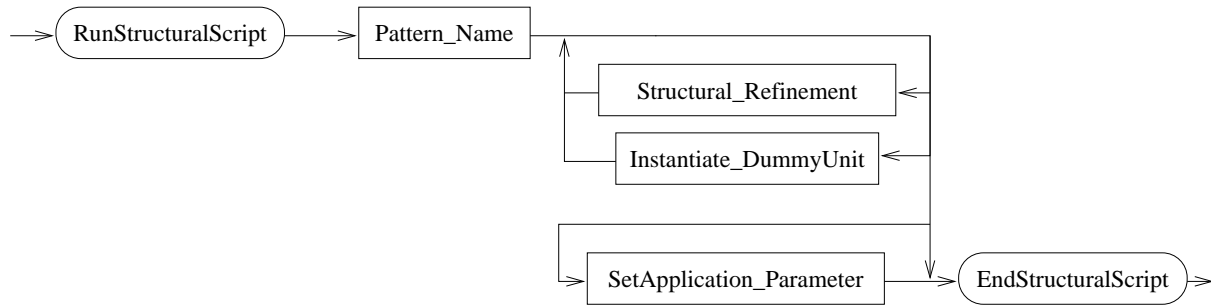


Figure 6.22: EBNF graph for the “Run_Structural_Script” nonterminal element in the EBNF graph in Figure 6.20.

As illustrated in this Figure, the user may define other structural operations like:

- component place-holder instantiation to tools from the Triana toolbox;
- parameterisation of these tools. The EBNF graph for this parameterisation is presented in Figure 6.23.

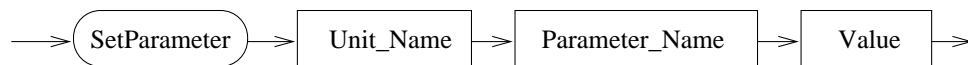


Figure 6.23: EBNF graph for the “SetApplication_Parameter” nonterminal element in the EBNF graph in Figure 6.22.

The following sub-script illustrates the instantiation of component place-holders as well as the parameterisation of a particular tool, namely the definition of the “filename” parameter in the “DataFrameReader” tool.

```

RunStructuralScript ImgProjection
  Decrease 1
  Instantiate DummyUnit
    /home/mcg/working/toolboxes/Demos/GalaxySim/DataFrameReader.xml
  SetParameter DataFrameReader fileName /home/mcg/working/triana/old_out.drt
  Instantiate DummyUnit1
    /home/mcg/working/toolboxes/Demos/GalaxySim/ViewPointProjection.xml
EndStructuralScript
  
```

Please note that the parameterisation of tools is also possible in a regular script, i.e. it is not restricted to sub-scripts.

Finally, it is worthwhile mentioning that, within a script, it is also possible to manipulate an already embedded pattern. For example, it is possible to define the following:

```
Create Pipeline ImgProjection
RunStructuralScript ImgProjection
  Decrease 1
  Instantiate DummyUnit
    /home/mcg/working/toolboxes/Demos/GalaxySim/DataFrameReader.xml
  SetParameter DataFrameReader fileName
    /home/mcg/working/triana/old_out.drt
EndStructuralScript
Embed ImgProjection DummyUnit
Instantiate ImgProjection.DummyUnit1
  /home/mcg/working/toolboxes/Demos/GalaxySim/ViewPointProjection.xml
```

In this case, the “DummyUnit1” component place-holder in the “ImgProjection” pipeline is instantiated to the “ViewPointProjection.xml” tool after the pipeline has been embedded. The access to that component place-holder is made through the identifier “ImgProjection.DummyUnit1”, i.e. this identifier results from the concatenation of the identifier of the pattern and the identifier of the component place-holder.

As previously described, the next step towards a pattern-based configuration is the definition of the data and control flows between components and subsequently their manipulation through Behavioural Operators. It is also the responsibility of the *pattern controller* within a pattern to enforce such behavioural patterns and give support to the execution of the Behavioural Operators. This is described in the next sub-section.

6.4.3 Execution Control from the Triana GUI and from Scripts

Workflow enactment in Triana defines that the execution of Units composing a workflow is triggered by the arrival of the sufficient data to that Unit. Namely, upon connecting the Units in a workflow, the user requests its whole execution through a *Run* control button in the GUI. As a result, the first Units to be triggered are the ones which are not dependent on data from other Units. Typically, such Units produce data that will be fed into other tools which are then rescheduled.

As previously described, each Unit’s implementation requires the definition of the type of the *input data nodes* in terms of being *mandatory/essential* or not. Each Unit’s execution is supported by a Triana *Task*, and if a Unit’s Task is already executing (e.g. through a control node named *Trigger node*), the existence of that mandatory node in the Unit implies the blocking of the Task when the node is read, until new data arrives in this node. Conversely, if the Task is not already executing, the arrival of data in a mandatory node may automatically trigger that Task’s execution (e.g. if the Unit/Task only has that mandatory input node, and all other input nodes are optional).

Such behaviour, provided by default in Triana, supports the *Streaming (Data-flow)* Behavioural Pattern. Consequently, the combination of this Behavioural Pattern with the defined Structural Patterns in an application's configuration is automatically available to the user. Many of the examples presented in Chapter 7 rely on this *Data-flow* Behavioural Pattern, since this pattern is commonly used in parallel and distributed systems.

While the default Triana's *Data-flow* Behavioural Pattern becomes very useful for many examples, it also restricted the implementation of alternative Behavioural Patterns. Namely, in the Triana version (3.1) used for the Pattern/Operator implementation, control flow is generally dependent on data flow, and the available separated control mechanisms provided by the *trigger nodes* do not support a powerful independent control-based execution flow. Namely, although *trigger nodes* can be used for control flow independently from the triggering mechanism associated to the mandatory data nodes, they also restrict a Task's execution. Specifically,

- A trigger node is only effective if it is declared as *mandatory/essential*. This means that a control message sent through a channel connected to a Task's trigger node is ignored if the trigger node is declared as *optional* (i.e. non-mandatory).
- An effective (i.e. essential) trigger node blocks the execution of a Task until it is sent a control message.

Consequently, only an essential trigger node may be used to trigger a Task's execution and it blocks that Task's execution until a control message is explicitly sent to that node. This means that a Task with two essential trigger nodes can only run if two control messages are sent, one to each trigger node.

Nevertheless, we used trigger nodes in our implementation in order to provide additional execution control among the tasks in a pattern-based workflow. A more detailed description of execution control will be described in a sub-section ahead. In the following, we describe the existing data and flow connections in a Pattern Instance in general.

Figure 6.24 presents one example of the connections supporting data and control flow within a particular Pattern Instance (PI) in the extended Triana. The represented Pattern Instance is a Star-based Hierarchical Pattern Instance named "Star" whose components are themselves PIs. Namely, the nucleus of the Star is a Pipeline-based PI named "ImgProjection", which is also presented in the Figure. The two satellites are two Pipeline-based PIs named "ImgProcessing" and "ImgAnalysis" (their detailed configuration is absent in the Figure).

The nucleus of the Star, i.e. *ImgProjection*, is connected to the satellites through output and input data nodes in each of the involved PIs. However, as represented in Figure 6.24 all components of the Star PI, i.e. the nucleus and the two satellites, are also connected to that PI's *pattern controller*, namely the *DrawStar* unit. These connections from the pattern controller bind to *trigger nodes*, one in each element in the Pattern Instance, through which the pattern controller may send control messages to all components in the PI.

In each PI in the extended Triana, may it be hierarchical or not, its pattern controller is always connected to all elements composing that PI through trigger nodes. The state of those individual trigger nodes may be toggled between *mandatory* and *optional* through the Triana nodes and also from scripts. Such will be described in a following sub-section.

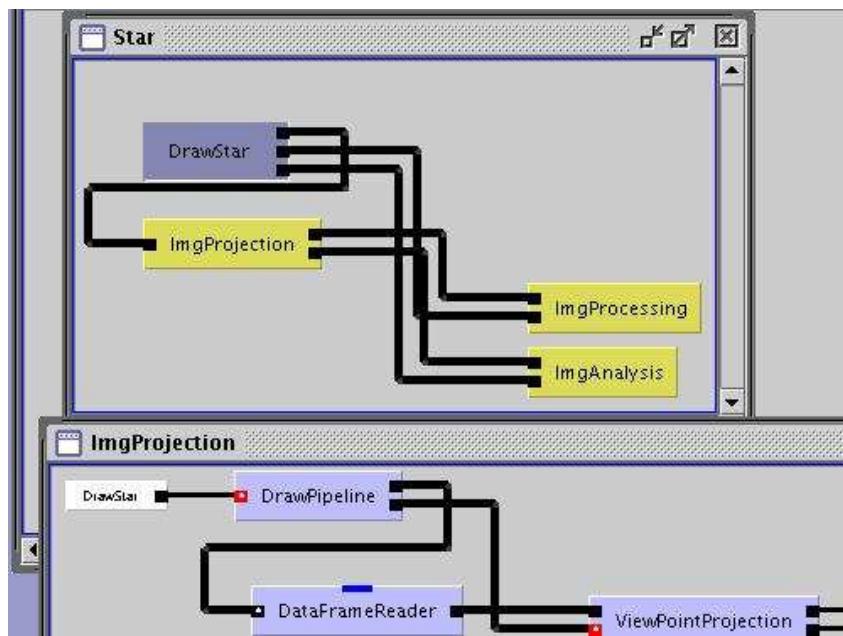


Figure 6.24: The data and control flow connections in a (Hierarchical) Pattern Instance.

Execution Operators

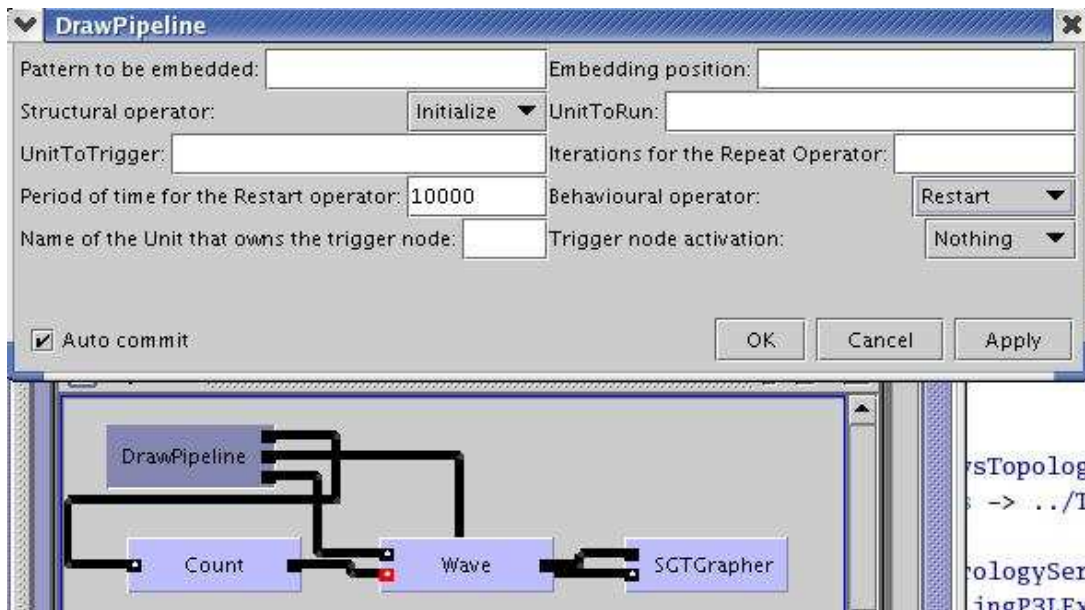


Figure 6.25: The parameter panel representing the Execution Operators and their arguments. The Restart Operator is selected to launch the periodic execution every 10000 milliseconds.

The implementation of the Behavioural Operators in Triana was restricted to the Execution Operators, namely, the *Start*, *Terminate*, *Restart*, *Repeat*, *Limit*, and a limited version of the *Stop* and *Resume* operators (e.g. the checkpointing of the pattern's state is not implemented yet). Similarly to the Structural Operators, the Execution Operators are accessible in the Triana GUI through the parameter panel of a Pattern Instance (PI) as presented in Figure 6.25. The operators are activated upon selection and as long as their necessary arguments are properly defined

through that panel.

For example, Figure 6.25 represents the parameter panel of a Pipeline PI where the *Restart* Operator is selected. The defined time period for this operator is 10000 milliseconds, as represented in the Figure.

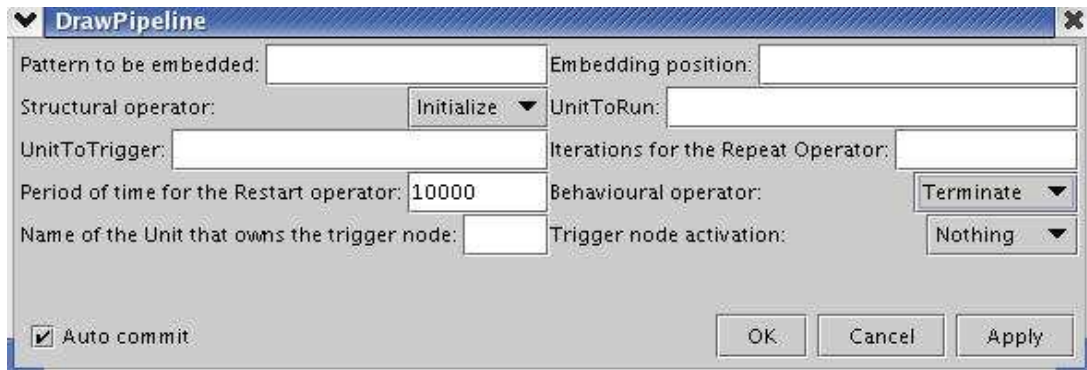


Figure 6.26: Application of the Terminate operator.

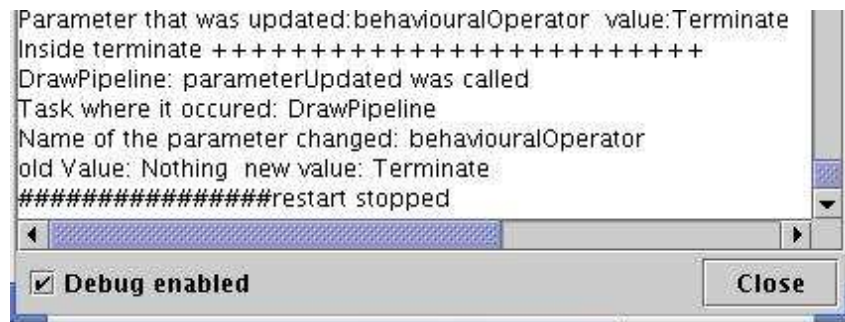


Figure 6.27: Execution debug information generated upon application of the Terminate operator to a pattern-based application ruled by the Restart operator.

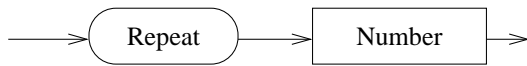
The described automatic re-execution can be stopped at any time by applying the *Terminate* Behavioural Operator as presented in Figures 6.26 and 6.27. In our Triana implementation, the *Terminate* not only aborts a pattern-based application's execution triggered by the *Start* operator, but also aborts the effect of the *Repeat* and *Restart* operators.

Also similarly to the Structural Operators, the implemented Execution Operators can be activated through a script. Therefore, scripts may either include only Structural Operators for application configuration, only Execution Operators for execution control, or both kinds of operators.

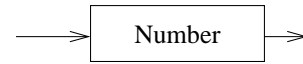
On one hand, the simplified semantics of the usage of individual Execution operators from a script is presented in Figure 6.28 (the *Terminate* operator was omitted). The usage of these operators from the Triana GUI, in particular, require the previous definition of their necessary parameters in the parameter panel.

On the other hand, Figure 6.29 presents the simplified graph of execution control through operators of a pattern-based application, both from scripts and from the Triana GUI. Except for the *Define_Execution_Control* non-terminal element in the EBNF graph, all other non-terminal elements in the graph in Figure 6.29 were previously defined in Figure 6.28.

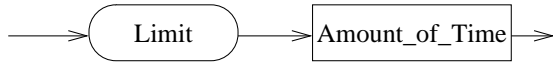
Repeat_Execution



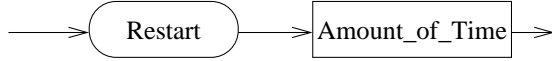
Amount_of_Time



Limit_Execution



Restart_Execution



Stop_Resume

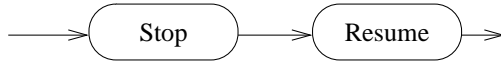


Figure 6.28: EBNF graph for the Execution Operators.

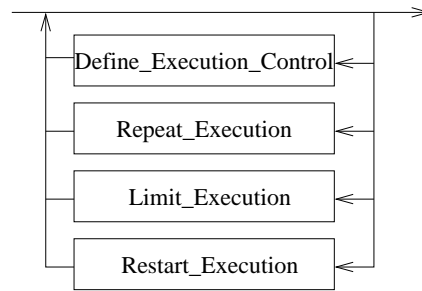


Figure 6.29: EBNF graph for the execution control of pattern-based applications.

The EBNF graph for *Define_Execution_Control*, in turn, is presented in Figure 6.30.

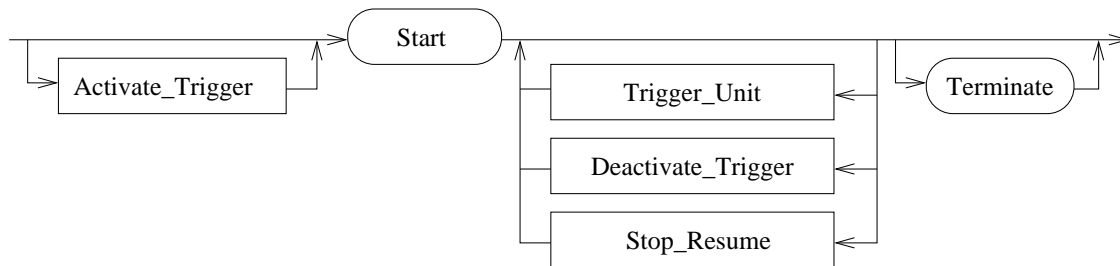


Figure 6.30: EBNF graph for explicit execution control including the usage of trigger nodes.

Specifically, Figure 6.30 represents the graph (both from a script and from the GUI) of:

- the invocation of the *Start*, *Stop*, *Resume*, and *Terminate* Execution Operators;
- the explicit flow control of an execution supported by Triana's *trigger nodes*, and its relation with the previous execution operators.

This explicit management of control flow within Patterns is described in the next-subsection, along with the definition of the non-terminal elements in the EBNF graph in Figure 6.30 which are related to *trigger nodes*.

Independent Manipulation of the Control Flow

As previously described, a trigger node in Triana may be defined as *mandatory/essential* or *optional*. Moreover, it is possible to change the state of a trigger node from *mandatory* to *optional*, and vice-versa. We define that

- a) an *optional trigger node* is in the *silent/non-active* state;
- b) an *essential trigger node* is in the *active* state.

In the silent state, a trigger node has no influence upon the execution, and the control is driven by Triana's data flow model previously cited. However, in the active state, execution control is stopped at the unit that owns that trigger node. This means that, although data may arrive in that unit's data nodes, only when the trigger node is "triggered" the execution flow is allowed to proceed.

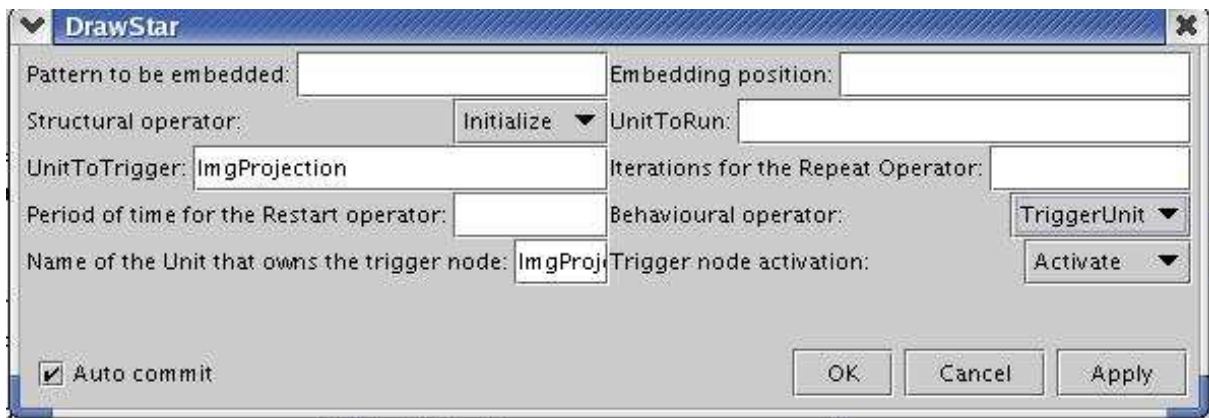


Figure 6.31: Activating a trigger node from a Pattern Instance's parameter panel.

The user can, at any time, activate and deactivate a trigger node through the operator panel of the Pattern Instance (i.e. the panel associated to the PI's pattern controller). This is illustrated in Figure 6.31. Specifically, the button "Trigger node activation" allows toggling the state of the trigger node identified by the parameter "Name of the Unit that owns the trigger node". In the figure, that particular trigger node was activated. Furthermore, Figure 6.31 also presents the selection of the *TriggerUnit* operation that results on the sending of a control message to the unit "ImgProjection" defined in the parameter "UnitToTrigger".

The following example also shows the usage of the control mechanisms provided by trigger nodes through a script. The line numbers in the script were introduced for their reference in the description text.

```
1: Initialize
2: Increase 1
3: Instantiate DummyUnit /home/mcg/working/toolboxes/Common/Const/ConstGen.xml
4: SetParameter ConstGen constant 7.0
5: Instantiate DummyUnit1 /home/mcg/working/toolboxes-dev/Patterns/Inc.xml
6: Instantiate DummyUnit2 /home/mcg/working/toolboxes-dev/Patterns/Inc.xml
```



```

7: Instantiate DummyUnit3 /home/mcg/working/toolboxes/Common/Const/ConstView.xml
8: Activate Inc
9: Activate ConstView
10: Start
11: TriggerUnit ConstView
12: TriggerUnit Inc

```

The script above is associated with a *DrawPipeline* unit and generates a Pipeline PT with four component place-holders (lines 1 and 2). In turn, lines 3 to 7 generate a Pattern Instance as a result of the instantiation of all component place-holders. In the pipeline, the value of a constant (generated by the “ConstGen” tool – lines 3 and 4) is to be incremented twice (through the “Inc” tool – lines 5 and 6) and displayed in the last stage (through the “ConstView” tool – line 7). Please note that in Triana, the creation of two instances of the “Inc” tool will generate two units named “Inc” (which instantiates the “DummyUnit1”) and “Inc2” (which instantiates “DummyUnit2”).

Additionally, lines 8 and 9 activate the trigger nodes of the tools “Inc” and “ConstView”. We recall that a Unit’s trigger node supports the connection to the pattern controller of a PI. Consequently, although the execution of the Pipeline PI is launched through the *Start* Execution Operator in line 10, the constant sent by the “ConstGen” tool does not automatically trigger the execution of the “Inc” tool, as would be the normal behaviour. Only when the trigger node of the “Inc” tool is sent a control message, what happens in line 12, the “Inc” tool’s execution is allowed to proceed. Such combination of the activation/de-activation of the trigger nodes together with the triggering control messages allows the emulation of the *Stop* and *Resume* operators.

The described control mechanisms together with the Execution Operators may also support the definition of different coordination schemes. This will be illustrated in an example in section 7.6.2 ahead.

6.4.4 Implementation in Triana

In this section we describe how Structural Pattern Templates, Structural Operators, and Behavioural Operators are implemented in the context of Triana’s class hierarchy (Triana version 3.1).

Intrinsically, a Pattern Template is a *group entity*: it contains a set of entities (connected in a pre-defined way) that is seen from the outside as a single entity representing and giving access to the elements in the set. As previously cited, Triana itself has the concept of a *group of units*:

- a) a group in Triana is a component that encapsulates a set of units;
- b) it has a recursive definition (it can contain other groups of units);
- c) a group owns a set of input and output ports to support the connection to the input and output ports of the encapsulated units.

As such, a Pattern Template was naturally represented as a group in Triana. In turn, a *Pattern Instance* is an already instantiated pattern template. Figure 6.32 describes, in a simplified

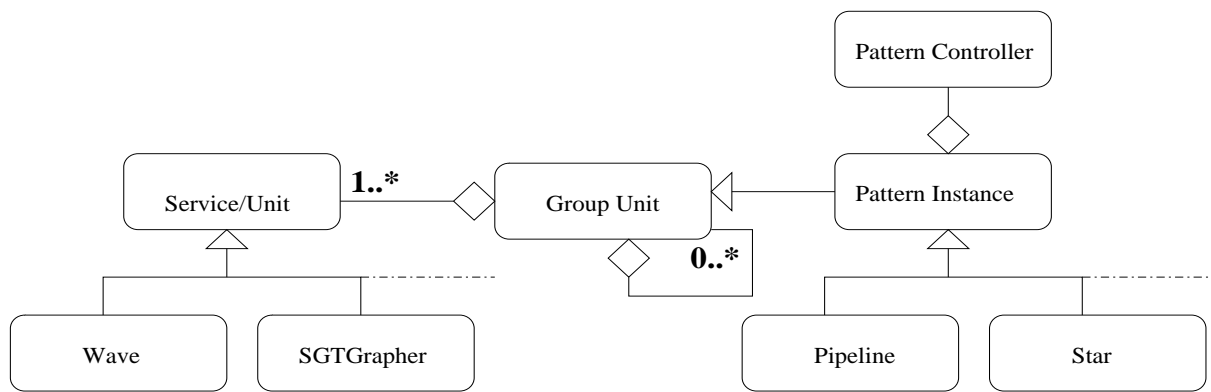


Figure 6.32: *Definition of a Pattern Instance.*

way, the meaning of a Pattern Instance in the context of Triana. Namely, a *Pattern Instance* is a *Group unit* (i.e. a *group of units*), which in turn contains Triana units (like *Wave*, etc) and may contain other *Group units* as well. Each particular *Pattern Instance* (e.g. *Pipeline*, *Facade*, etc) aggregates units according to a specific structure. Finally, all *Pattern Instances* own a *Pattern Controller* which is responsible for:

- a) keeping track of the elements (units) in the pattern and how they should be connected;
- b) implementing the Structural Operators which act upon the Structural Pattern;
- c) implementing the Behavioural Operators which act upon the Structural Pattern combined with some Behavioural Pattern;
- d) implementing a small script engine which evaluates Structural and Behavioural Operators read from a file;
- e) detecting and processing relevant events, such as a request to instantiate a *DummyUnit*, or a notification of the end of the execution of units within the Pattern Instance. For instance, this notification allows the *Pattern Controller* to evaluate if the pattern, as a whole, has finished its execution.

A Few Triana Classes

Figure 6.33 shows a very small and simplified subset of Triana's Class Hierarchy, specifically, some of the classes which are in some way directly related to the implementation of the Pattern Templates and the execution control of their associated Pattern Instances. The Figure just aims at providing a general overview of some of the more important entities, and in fact, it is not completely accurate: the interface hierarchy was omitted, and the name of some classes was replaced with the name of one of the interfaces they implement (this was done for the *Tool*, *Task*, and *TaskGraph* entities).

Figure 6.33 is divided into three major areas: *Unit definition* presents the basic class for defining a Triana service/unit; *Helper entities* shows some essential classes for helping realizing the execution of a unit; and finally, *GUI entities* gives a small example of the classes that support the interaction with the users.

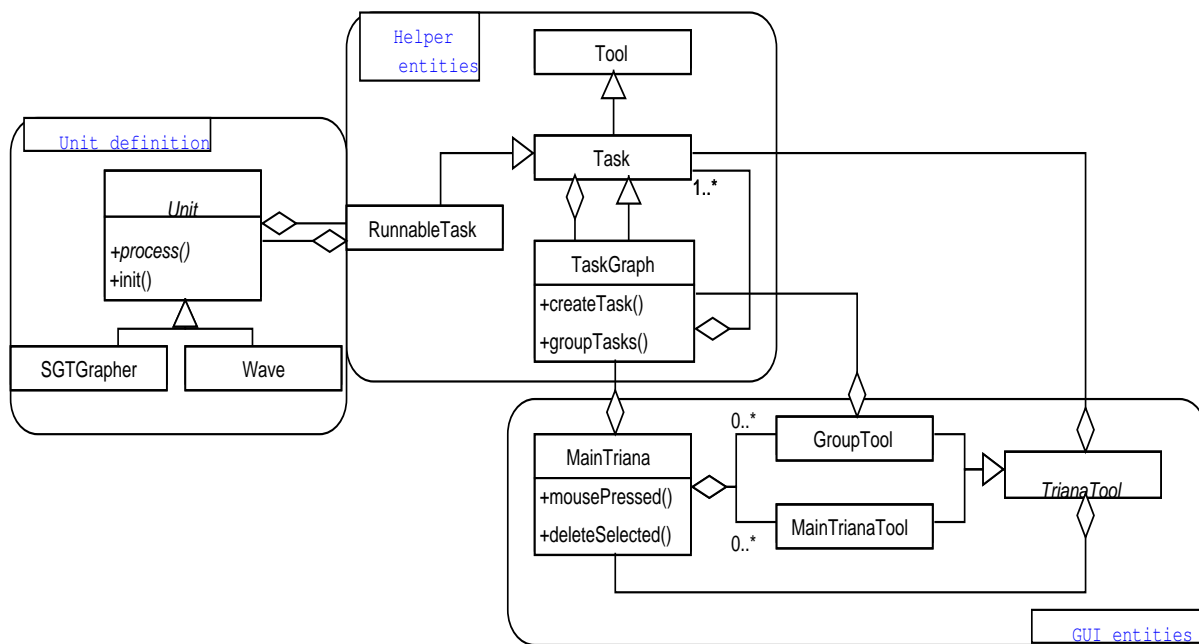


Figure 6.33: UML simplified description of some Triana classes.

A new service is defined by extending the *Unit* abstract class, which provides all the necessary methods. Some of these methods have to be explicitly implemented by the service developers, and others are optional. For example, the *process()* method is mandatory and specifies the unit's specific actions. In an optional method like *init()*, the user may define the initialisation actions that have to be done before the specific service code executes (i.e. before *process()* is executed).

Triana also provides an event mechanism through parameter definition: users may define parameters to the unit, and whenever the value of a parameter changes an event is generated, to be caught by the entities declared as "listeners" to that parameter.

Helper classes like *Tool*, *Task*, and *RunnableTask*, provide the necessary code to execute a unit, and to send and receive data through nodes. *Tool* defines code common to all units in the toolbox (e.g. parameter management code, like code to get the name of all the service's parameters). A tool object results from the evaluation of a unit's XML file. *Task* extends *Tool* and represents a task in a task-graph, i.e. an entity that can be connected to other entities forming a data-flow network. *RunnableTask* makes the connection between a *Unit* and a *Task*. It initialises an associated unit (e.g. an object of class *Wave*) by calling its *init()* method. Furthermore, it implements the data handling capability of a *Task* (e.g. keeps track of which nodes have data that has not yet been processed, and wakes up a task when there is data ready to be read in all input nodes).

As represented in Figure 6.33, a *TaskGraph* is itself a *Task* which provides code to represent a group of tasks. In Triana, a taskgraph contains a collection of tasks linked by cables. Whenever a task is created, it is always created in the context of a taskgraph (and contains a reference to this taskgraph as shown in Figure 6.33).

A *TaskGraph* provides methods for: creating a new task within it (*createTask()*); connect-

ing/disconnecting tasks which belong to the taskgraph; creating a new sub-taskgraph out of a group of tasks that belong to the taskgraph (*groupTasks()*); etc.

MainTriana handles the user interface for interconnecting units – it represents an area where the tasks belonging to a (single) taskgraph are drawn. The icon representing a task is associated with *MainTrianaTool* and the icon representing a (sub-)taskgraph is associated with *GroupTool*. *MainTriana* gives support to several actions like: evaluating if two nodes belonging to two components (i.e. icons) are compatible; connecting compatible components' nodes through a cable; drawing pop-up menus; moving selected units; grouping selected units; etc.

Some of the described classes communicate through an event mechanism. For example a taskgraph is a listener to the tasks that it represents (e.g. it gets notified when a task is disconnected from another task). Besides tasks, it is possible to listen to events from task's nodes, cables, taskgraphs, parameter updates, etc.

Implementation of Patterns and Operators

One of the restrictions concerning the implementation of the Structural Patterns and Operators was that the Triana's code should be changed as least as possible, in order to keep our development independent from Triana's internal changes. As such, the simplest solution was to implement Patterns (and Operators) as *Unit* extensions.

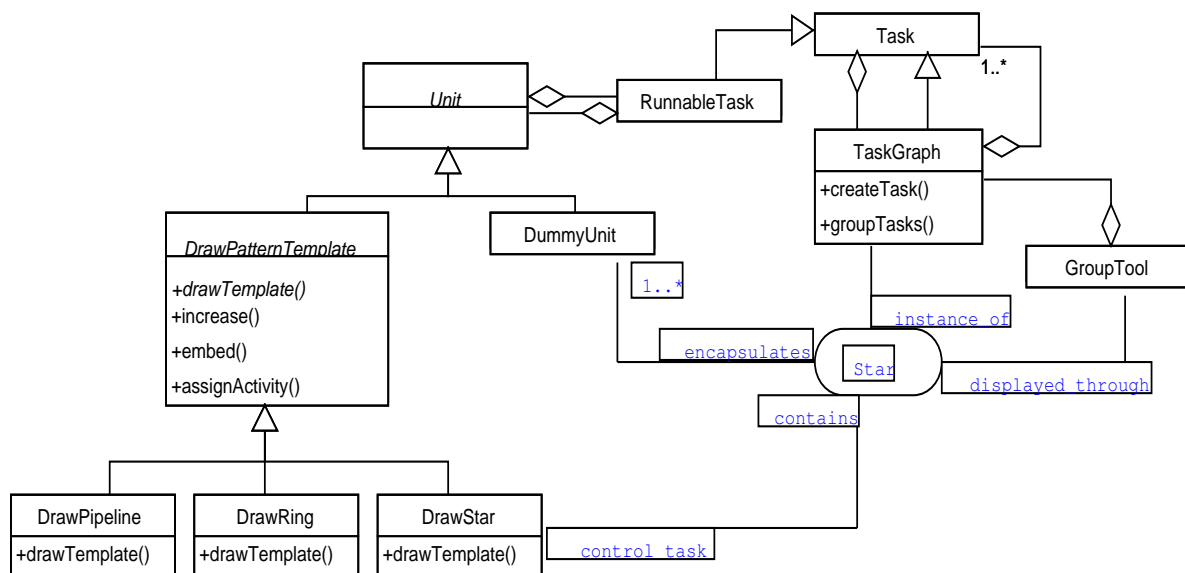


Figure 6.34: Simplified UML definition of the classes for creating and manipulating Pattern Templates and Instances through Structural and Behavioural Operators, respectively. The definition includes a particular example of the Ring pattern template.

As shown in Figure 6.34, class *DrawPatternTemplate* is an abstract class that extends *Unit* and defines the common code for the construction of Pattern Instances and their execution control. The existing abstract methods are redefined by each of the specific sub-classes like

DrawPipeline, *DrawStar*, and *DrawRing*. These are the *pattern controllers* of the patterns they represent.

Through the association with *RunnableTask*, each of these sub-classes has access to its associated task and then to the taskgraph to which it belongs. Consequently, the units can create new tasks and new taskgraphs by invoking *taskgraph.createTask()* and *taskgraph.groupTasks()*, respectively.

For example, to create a Pattern Template (and subsequently a PI) like *Star* in Figure 6.34, an instance of *DrawStar* (*DrawStar*) creates a new taskgraph that includes the instance itself. This *DrawStar*'s instance will act as a *pattern controller* for the pattern represented by the newly created taskgraph. The major supported functionalities supported by this pattern controller are described in the following.

- The first responsibility of the pattern controller is to draw the specific (star) structure by creating a default set of component place holders, i.e. *DummyUnit* tasks, inside the taskgraph, and by connecting them in the shape of a star. A *DummyUnit* instance provides an initialisation parameter for the selection of a specific unit from the toolbox that will instantiate that component place holder. Moreover, all *DummyUnits* are also connected to the pattern controller through trigger nodes.
- The second responsibility is to catch relevant events at *DummyUnits*' level. For example, the instantiation of the *DummyUnit* is in fact implemented by the *DrawStar*. *DrawStar* is declared as a "listener" to the parameter for *DummyUnit* initialisation, and wakes up as soon as this parameter changes. It then replaces the *DummyUnit* with a specific service from the toolbox, keeping the existing connection through a trigger node. The toggling of the state of these trigger nodes is another example of the events processed by the pattern controller.
- The third responsibility is to implement the Structural Operators like *Increase*, *Embed*, etc. The operators' major code is defined in *DrawPatternTemplate*, and specific actions are left to the subtypes.

For example, the *Increase* operator needs two actions:

- a) to create and draw a new *DummyUnit* element;
- b) to identify the connection element, i.e. to which of the already existing *DummyUnits* should the new element be attached to.

Action a) is common to all sub-types, so it is implemented in the *DrawPatternTemplate* class. Action b), in turn, is specific of each Structural Pattern. For example, for a *Star* Pattern Template a new *DummyUnit* has to be attached to the *nucleus* of the star, whereas for the pipeline a new *DummyUnit* is added to one of the ends of the pipeline. As such, each subtype redefines the abstract method *getConnectionElement()* which identifies the *DummyUnit* that represents the adequate connection point.

- The fourth responsibility is to support the execution data and control flows within its own Pattern Instance, but also in the context of a Hierarchic Pattern Instance it may

belong to. To this extent, all pattern controllers within a Hierarchic Pattern Instance are connected to the pattern controller of the enclosing Pattern Instance.

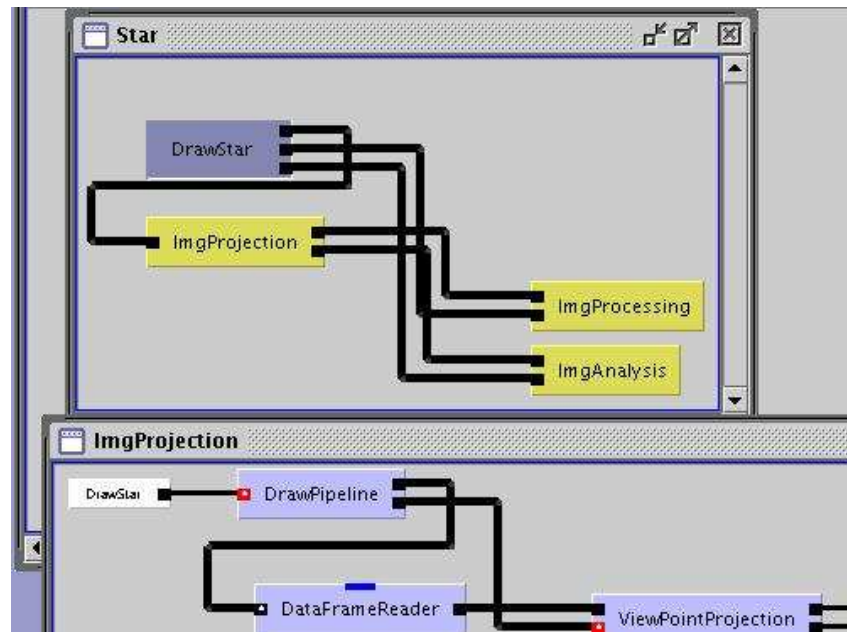


Figure 6.35: The data and control flow connections in a (Hierarchical) Pattern Instance.

For example, in the example in Figure 6.35 all *DrawPipeline* pattern controllers of the pipeline-based PIs “*ImgProjection*”, “*ImgProcessing*”, and “*ImgAnalysis*”, are connected to the *DrawStar* pattern controller of the enclosing star-based PI. Through these connections, the *DrawStar* pattern controller may for example send control messages to the pattern controllers of the embedded patterns.

- Finally, the pattern controller is also responsible for the execution support of the implemented Execution Operators within the context of its own PI. For example, to support the *Repeat* operator applied to its own PI, the pattern controller has to detect
 - a) when all tasks within the PI have terminated executing before relaunching the PI’s execution for the next *Repeat* iteration;
 - b) if number of the desired repeated invocations (as defined in the parameter for the *Repeat* operator) has been reached or not. In case all iterations have been accomplished, the PI’s execution is not triggered again.

Triana proved to be an adequate environment to support the implementation of Patterns and Operators. However, the version used for our implementation had some limitations on the propagation of some necessary events for the distributed execution of a Hierarchical Pattern Instance.

In the following section we also describe how to map some of the described Execution Operators to a particular distributed resource manager for execution control.

6.5 Mapping to the DRMAA API

The Triana supporting architecture already provides the mapping to several APIs for distributed environments, as previously described in section 6.3.1. However, the proposed Behavioural Operators require a set of functionalities allowing the fine tuning of the execution control of jobs. For this reason, we highlight the relevance of the *Distributed Resource Management Application API (DRMAA)* [43] API, which provides a standard job control programming interface and allows a distributed representation of the application tasks. Namely, the execution control primitives provide us with operations which are suitable for implementation of the Behavioural Operators proposed in our model. Furthermore, the relevance of the DRMAA API as a specification for submission, control, and monitoring of distributed jobs has been supported by an increasing number of systems whose implementations are conform to the DRMAA API [263–265].

The DRMAA specification allows the submission and control of *jobs* to one or more *distributed resource management systems (DRMSs)*. Since DRMAA abstracts fundamental job interfaces of DRMSs, it facilitates integration of application programs. Concretely, a *job* is a running application on a DRMS and it is identified by a *job_id* attribute that is passed back by the DRMS upon job submission. This attribute is used by the functions that support job control and monitoring, e.g. termination and suspension operations.

DRMAA uses an IDL [108]-like definition (with *IN*, *OUT* and *INOUT* parameters) for specifying the API, and also provides support for handling errors (via error codes).

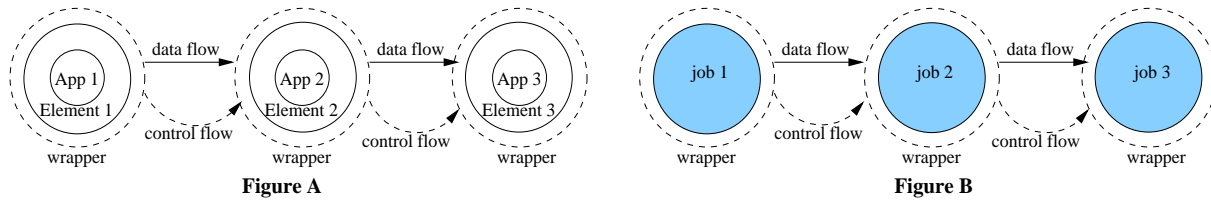


Figure 6.36: A Streaming (data-flow) Behavioural Pattern combined with a Pipeline Structural Pattern. Figure A shows the entities before the execution of the Pattern Instance. Figure B shows the jobs created by a DRMS to support the execution of the applications (“App1”.. “App3”) in the “Pipeline” Pattern Instance.

The mapping of the Execution Operators to the DRMAA API is illustrated through a simple application example. This example is configured as a three stage *Pipeline Structural Pattern* combined with the *Streaming (data-flow) Behavioural Pattern*, and instantiated to executable applications (e.g. representing tools/services) named *App1*, *App2*, and *App3*, as represented in the left side of Figure 6.36 (i.e. “Figure A”). The resulting Pattern Instance is simply designated as *Pipeline*. Such example is similar to the one used in the semantic description of the Execution Operators in section 4.4, as well as to the example presented in Figure 6.2 in section 6.2.1.

The execution of the applications *App1* .. *App3* using the DRMAA specification requires the definition of some attributes like the application’s name, its initial input parameters, the necessary remote environment that has to be set up for the application to run, and so forth. These attributes are used to explicitly configure the task to be run in a resource manager. We

designate as *Element* the entity that represents an application and its attributes which are necessary to run the application in a DRMS. In the description of the mapping of the Behavioural Operators the initialisation of an application's attributes is most times omitted. Only relevant attributes are explicitly initialised.

In Figure 6.36, "Figure A" represents the *Elements* encapsulating the executable applications. In turn, the *Elements* are themselves encapsulated in *wrappers* that enforce the *Streaming (data-flow)* Behavioural Pattern jointly with the *Pattern Controller*, as previously described.

The execution of the *Elements* in a DRSM are supported by *Jobs* as represented in the right side of Figure 6.36, i.e. "Figure B". It is assumed that the standard output of *Element 1* is redirected to the standard input of *Element 2*, and the standard output of *Element 2* is in turn redirected to the standard input of *Element 3*. One way to map this redirection to the DRMAA is to define the parameters *drmaa_input_path* and *drmaa_output_path* for the jobs that support the execution of the *Elements*.

The DRMAA specification has the notion of sessions. However, in version 1.0 only one session can be open at a time, meaning that the nesting of sessions is not supported. For simplification reasons, it is assumed a single DRMAA session for all the operators. It is therefore assumed that the DRMAA's initialisation (i.e. *drmaa_init*) and exit (i.e. *drmaa_exit*) routines are called, respectively, after the Pattern Instance is created and in the end of the script program. The alternative would be to create a new DRMAA session (with *drmaa_init*) in the beginning of each Behavioural Operator's definition, and terminate that session (with *drmaa_exit*) at the end of the operators' definition.

A few more assumptions are made:

- a) A Pattern Instance has an object associated with it. The Object gives access to some variables like:

Element pattern_elements[*MAX_ELEMS*] This vector contains the *Elements* that compose a specific pattern instance.

String job_identifiers[*MAX_ELEMS*] This vector represents the job identifiers returned by the *drmaa_run_job* routine for the jobs that are created to support the activities represented in the vector *pattern_elements*. The order of the activities is preserved, i.e. the first job identifier in the vector *pattern_elements* belongs to the first job identifier in the vector *job_identifiers*.

- b) DRMAA variables frequently used:

INOUT jt Represents the job template (opaque handle).

INOUT drmaa_context_error_buf Contains a context-sensitive error upon failed return.

- c) Error processing is simplified. It uses the auxiliary function:

process_error(IN ret, IN drmaa_context_error_buf) This simplified routine is used to check if the result of the last call to a DRMAA routine (the result is passed in *ret*) is different from *DRMAA_ERRNO_SUCCESS*. If it is, it prints the error returned in *drmaa_context_error_buf* through the *drmaa_strerror* routine.

d) Other auxiliary routines:

define_attributes(IN jt, IN Element) This routine sets attributes of the job template *jt* based on the properties of a specific *Element*. For example, the following actions could be done inside the *define_attribute* routine:

```
ret = drmaa_set_attribute( jt, drmaa_remote_command,  
                          Element.executableName,  
                          drmaa_context_error_buf );  
process_error( ret, drmaa_context_error_buf );  
ret = drmaa_set_attribute( jt, drmaa_v_argv,  
                          Element.arguments,  
                          drmaa_context_error_buf );  
process_error( ret, drmaa_context_error_buf );  
...
```

Other attributes to be defined depend on the specific application to be run (which is accessed through *Element*).

The following sub-sections define the mappings of the *Start*, *Terminate*, *Stop*, *Resume*, *Restart*, *Repeat*, and *Limit* Execution Operators, and in the context of the simple application example previously described. It is worthwhile mentioning here that the Execution Operators are assumed to be executed sequentially due to the lack of adequate (workflow) constructs in the DRMAA.

6.5.1 Start and Terminate Behavioural Operators

The order by which the *Elements* are started in the Pipeline is from last to first. In this way, the first element to run, i.e. *Element 3*, will block waiting for data. Conversely, the first element to be terminated is the first one, followed by the *Element 2*, and *Element 3*.

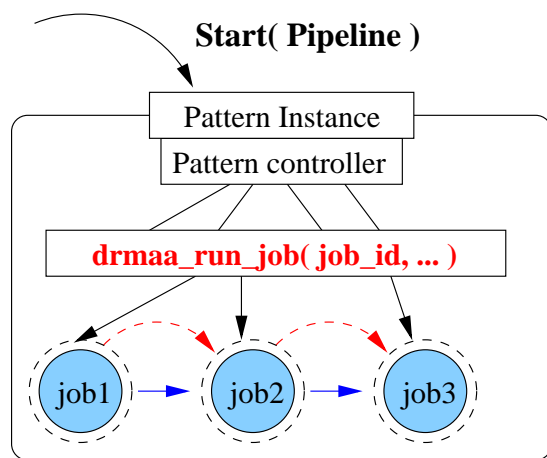


Figure 6.37: DRMAA mapping of the Start operator.

Start(Pipeline) (Figure 6.37):

```
/* The variable Pipeline.pattern_elements[0] represents the
  ``Element 1`` in the pipeline.
  */
for( int index = Pipeline.pattern_elements.length -1 ; index >= 0;
index -- ) { // launch all activities in the pipeline
    int ret;
    ret = drmaa_allocate_job_template( jt, drmaa_context_error_buf );
    process_error( ret, drmaa_context_error_buf );
    define_attributes( jt, Pipeline.pattern_elements[index] );
    /* As an example, it is also possible to define exactly the
    time at which all jobs in the pipeline will be started. The
    variable ``Pipeline.startTime`` defines the time at which all
    elements in the pipeline instance should start running.
    */
    ret = drmaa_set_attribute( jt, drmaa_start_time,
                              Pipeline.startTime,
                              drmaa_context_error_buf );
    process_error( ret, drmaa_context_error_buf );
    /* Now, it is necessary to run the job. The difference between
    running a single job or a bulk of jobs is again dependent on the
    ``Element`` that instantiates the pattern template
    (which is accessed through ``Pipeline.pattern_elements[index]``).
    For simplification, we assume that a single job is run.
    */
    ret = drmaa_run_job( job_id, jt, drmaa_context_error_buf );
    process_error( ret, drmaa_context_error_buf );
    /* Now, the job identifier is saved to allow access, later on, to
    the jobs belonging to this pipeline, from other Behavioural Patterns.
    */
    Pipeline.job_identifiers[index] = job_id;
} // end cycle for
```

Terminate(Pipeline) (Figure 6.38)

```
for( int index = 0; index < Pipeline.job_identifiers.length ; index++
) { // terminate all activities in the pipeline
    ret = drmaa_control( Pipeline.job_identifiers[index],
                        DRMAA_CONTROL_TERMINATE,
                        drmaa_context_error_buf );
    process_error( ret, drmaa_context_error_buf );
} \\ end cycle for
```

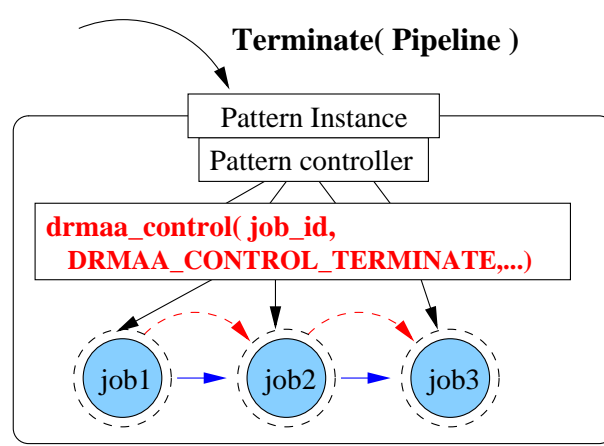


Figure 6.38: DRMAA mapping of the Terminate operator.

6.5.2 Stop and Resume Behavioural Operators

The first element to be stopped is *Element 1* and the last will be *Element 3*. For the *Resume* operator is the opposite: the first element to be resumed is *Element 3*.

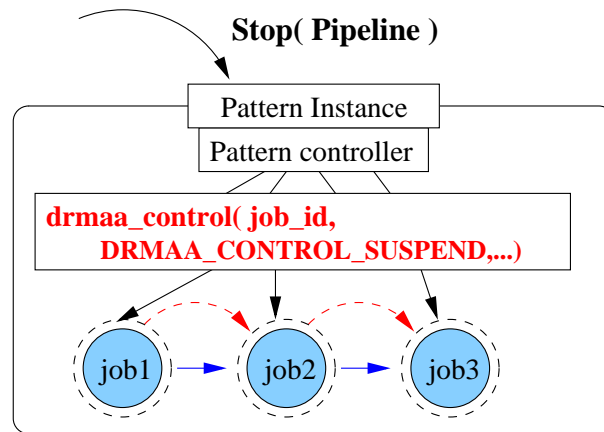


Figure 6.39: DRMAA mapping of the Stop operator.

Stop(Pipeline) (Figure 6.39):

```
for( int index = 0; index < Pipeline.job_identifiers.length ; index++
) { // suspend all activities in the pipeline
    ret = drmaa_control( Pipeline.job_identifiers[index],
                        DRMAA_CONTROL_SUSPEND,
                        drmaa_context_error_buf );
    process_error( ret, drmaa_context_error_buf );
} \\ end cycle for
```

Resume(Pipeline) (Figure 6.40):

```
for( int index = Pipeline.job_identifiers.length -1 ; index >= 0;
index --) { // resume all activities in the pipeline from the point
```

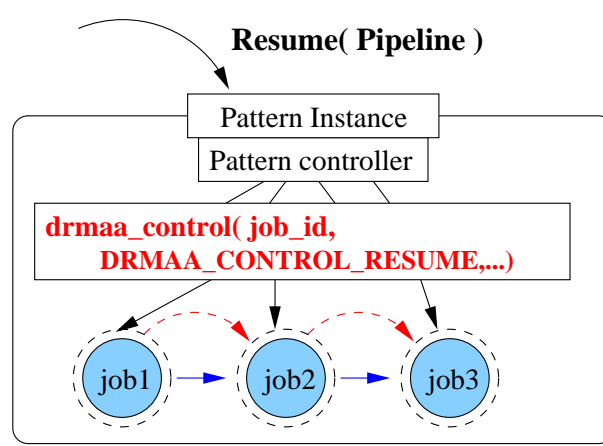


Figure 6.40: DRMAA mapping of the Resume operator.

```
// where they were suspended
ret = drmaa_control( Pipeline.job_identifiers[index],
                    DRMAA_CONTROL_RESUME,
                    drmaa_context_error_buf );
process_error( ret, drmaa_context_error_buf );
} \\ end cycle for
```

6.5.3 Restart and Repeat Behavioural Operators

The *Restart* operator defines a periodic re-start of the execution of a pattern, and the mapping uses the one defined above for the *Start* operator. However, it is assumed that the time the applications composing the pipeline instance take to run is less than the period of time that is passed as argument to the *Restart* operator. Moreover, the *Restart* operator is assumed to be endless. Although not represented, the *TerminateRestart* could be implemented by changing the value of a variable to be checked by the *Restart* operator prior re-calling the *Start* operation, similarly to the semantics description in Figure 4.41 in section 4.4.6³. Meanwhile, the invocation of the *Terminate* operator would only abort the current execution of the “Pipeline” as a result of the invocation of `drmaa_control(job_id, DRMAA_CONTROL_TERMINATE,...)`.

Restart(time_period, Pipeline):

```
/* For simplification reasons, it is used a Unix-like ``alarm''
routine that generates an interruption when the timeout expires.
Similarly, it is assumed the existence of a ``pause'' routine
to block the process running the ``restart'' operator.
*/
for( ; ; )
{
    /* It is assumed that the ``Restart'' operator is endless.
    */
```

³Alternatively, assuming that the *Restart* operator can be aborted by a signal, such signal would be sent upon invocation of the *TerminateRestart* operator.

```

Start( Pipeline );
alarm( time_period );
pause();
}

```

As for the *Repeat* operator, it controls the number of consecutive times a pattern is to be executed. The operator guarantees that, for each individual pattern's execution, only when all applications in the pattern finish running, a new pattern's execution is then launched. The DRMAA API provides the *drmaa_synchronize* routine which supports this semantics since, upon invocation of the routine, the execution is only allowed to proceed when all the jobs passed as argument terminate their execution.

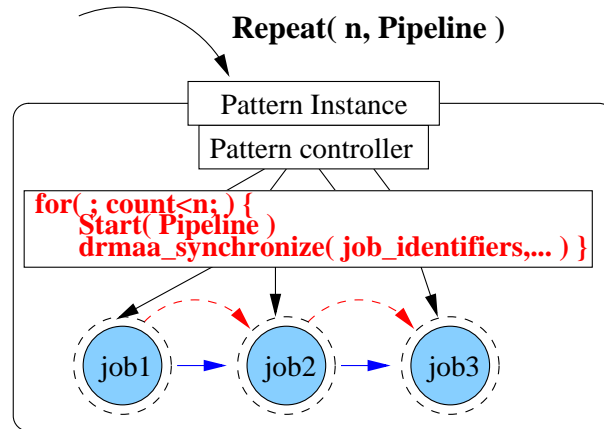


Figure 6.41: DRMAA mapping of the *Repeat* operator.

In the mapping of the *Repeat* operator bellow, it is defined that an individual execution of the pipeline pattern as a whole is supported by the *Start(Pipeline)* operator). The identifiers of the jobs that represent the applications forming the stages of that pipeline are then passed as the first argument to the *drmaa_synchronize*. Consequently, only when all those pipeline's jobs terminate, a new pipeline's execution is allowed. The *drmaa_synchronize* routine accepts a second argument which defines for how long such barrier-like synchronisation will hold. In this case, it is simply assumed that the value passed as argument (i.e. "timeout") is greater than the time all individual jobs will take to execution.

Repeat(n, Pipeline) (Figure 6.42):

```

for( int count = 0; count < n; count++ ) {
    Start( Pipeline );
    /* The timeout argument is assumed to be large enough to
       allow all jobs in the pipeline to terminate.
    */
    drmaa_synchronize( Pipeline.job_identifiers, timeout, 0,
                       drmaa_context_error_buf );
}

```

6.5.4 Limit Behavioural Pattern

The semantics of the *Limit(time_period,Pattern)* operator in section 4.4.5 specifies that the Pattern Instance this operator is applied to is already executing. The *Limit* operator just defines how much time the pattern still has left to run. To this extent, the implementation of the mapping to the DRMAA invokes the *drmaa_synchronize* routine using the “time_period” value as its second argument. This defines for how long to wait for all pipeline’s jobs to terminate. In case that timeout expires, it means that the pipeline’s execution as a whole has not finished yet and, therefore, it is necessary to explicitly abort its execution by invoking the *Terminate(Pipeline)* operator.

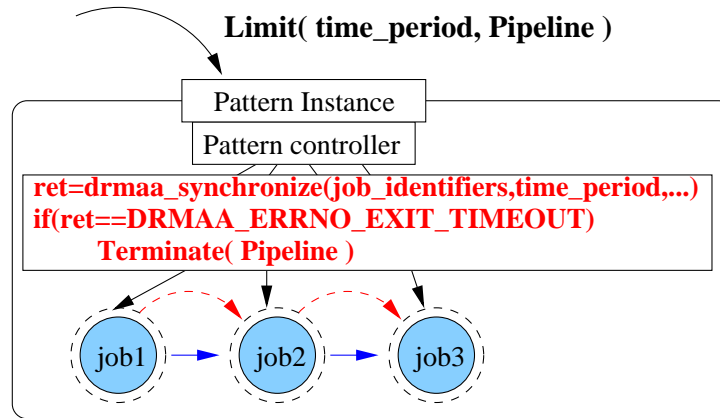


Figure 6.42: DRMAA mapping of the *Limit* operator.

Limit(time_period, Pipeline) (Figure 6.42):

```
/* The ``Limit`` operator waits that the jobs in the Pipeline
   terminate. In case the ``time\_period`` expires, the operator cancels
   the execution of all jobs in the Pipeline.
*/
ret = drmaa_synchronize( Pipeline.job_identifiers, time_period, 0,
                        drmaa_context_error_buf );
if( ret == DRMAA_ERRNO_EXIT_TIMEOUT )
    Terminate( Pipeline );
```

To consider the execution time limit from the moment the Pipeline Pattern Instance starts executing, the user may apply the compound operator *Limit(time_interval, Start(Pipeline))*, as discussed in section 4.5.1. The following description defines how that compound operator may be directly mapped to the DRMAA. Namely, the implementation is similar to the one for the *Start* operator, but it uses the *drmaa_wct_hlimit* DRMAA attribute to limit the Pipeline’s time of execution.

Limit(time_period, Start(Pipeline)):

```
/* The mapping of this compound operator is similar to the ``Start``
   operator. The only difference is that a special attribute limiting
   the execution time has to be set for the jobs.
```



```

*/
for( int index = Pipeline.pattern_elements.length -1 ; index >= 0;
index -- ) { // launch all activities in the pipeline
    int ret;
    ret = drmaa_allocate_job_template( jt, drmaa_context_error_buf );
    process_error( ret, drmaa_context_error_buf );
    define_attributes( jt, Pipeline.pattern_elements[index] );
    /* Definition of the limit of time to run the job: */
    ret = drmaa_set_attribute( jt, drmaa_wct_hlimit, time_period,
                             drmaa_context_error_buf );
    process_error( ret, drmaa_context_error_buf );
    ret = drmaa_run_job( job_id, jt, drmaa_context_error_buf );
    process_error( ret, drmaa_context_error_buf );
    Pipeline.job_identifiers[index] = job_id;
} // end cycle for

```

6.6 Summary

In this chapter we described the partial implementation of our model in a Grid-aware development environment.

The chapter defined a general architecture which is necessary to support the execution of our model and associated methodology over the Grid environment. This architecture highlights the necessity of diverse features to support a Pattern-based manipulation through Operators at different stages of the application development cycle. For instance:

- an adequate composition environment for application configuration;
- an API for distributed execution in Grid environments which may support the orchestration of diverse distributed applications/services;
- support of fine tuned control over the distributed execution, for instance to suspend and checkpoint the state of a distributed application so that its execution may be later resumed (e.g. [209]).

Additionally, the chapter presented the selected implementation platform, the Triana Problem Solving Environment, and discussed its adequacy for supporting the implementation of Patterns and Operators. Due to the complexity of our proposed model, only a subset of the previously discussed Patterns and Operators were effectively implemented within Triana.

This extension of Triana was discussed in terms of the usage of Patterns and Operators and the way they were implemented. Through the extended Triana, the user may configure and execute a pattern-based application according to the basic methodology as described in section 3.1.3. Due to the complex interactions underlying the dynamic reconfiguration characteristics discussed in 5.3, it was not possible to include them in the Triana implementation.

Finally, the chapter ends by describing a possible mapping of Behavioural Operators over a distributed resource manager system, namely the DRMAA specification. This was intended as a feasibility study of the mapping between part of the proposed model over a standardized API.

7

Validation

Contents

7.1	Introduction	220
7.2	Configuring Distributed Systems Topologies	221
7.3	Configuring a Problem Solving Environment	226
7.4	Skeleton Modelling	244
7.5	Analysis of Gravitational Waves	259
7.6	Galaxy Formation Example	269
7.7	Simulating Flexible Information Retrieval and Processing	281
7.8	Summary	288

This chapter illustrates, through examples, the expressiveness of the model. Some examples were tested using the extensions made to Triana workflow system, a Grid-aware Problem Solving Environment extended with Structural and Behavioural Patterns and Operators. Some other examples making use of pattern operators at the conceptual level are also included, in order to clarify the potentialities of the model.

7.1 Introduction

The purpose of this chapter is to exemplify how the pattern/operator model presented in this thesis can be used to build typical application configurations in distributed and Grid environments.

The enumerated examples may be divided in two groups. The first group includes three cases (sections 7.2, 7.3, and 7.4) that concern the application of the model at the conceptual level. Although these examples use some model features which are not actually implemented, it is our intention to further clarify the relevance of those features. The second group is a set of three examples (sections 7.5, 7.6, and 7.6) implemented over the Triana extension with patterns and operators as described in Chapter 6.

7.1.1 Conceptual Examples

The first conceptual example (section 7.2) highlights the fact that the proposed Structural Patterns are adequate to configure typical distributed systems topologies. The possibility of combining (the selected) Structural Patterns with different Behavioural Patterns allows not only to represent the typical interactions in those distributed systems topologies, but also to define other dissimilar data and control flows upon the same structures.

The second example (section 7.3) discusses the methodology for building a Problem Solving Environment which represents a common configuration in several areas. This example aims to highlight the relevance of the patterns we chose to be presented in this work, and the way they can be manipulated during the entire PSE development cycle, namely, from application configuration to execution control, and also towards reconfiguration.

The third example (section 7.4) makes a parallel between the patterns in the model, and similar abstractions for application configuration that have been ported to Grid environments, namely the *skeletons* programming abstractions. The example aims to clarify how patterns may represent skeletons, with the advantage that patterns may be manipulated for execution control and reconfiguration. Moreover, patterns include design concepts which are not covered by the skeleton definition.

7.1.2 Examples in Triana

The fourth example (section 7.5) defines a simulation in Triana for the analysis of Gravitational Waves. This example illustrates the usage of the implemented Patterns and Operators in the Triana environment both from the GUI and also from scripts. Specifically, the example demonstrates in general how to configure an application in the extended Triana and how to control its execution.

The fifth case (section 7.6) presents a simulation of a real-world application in Grid environments, specifically in the Astrophysics scientific domain. Concretely, it is illustrated how to configure a Galaxy formation example by using Pattern manipulation through Pattern Operators. The example was also developed both at the Triana GUI as well from pattern and operator scripts. Some scenarios for the Galaxy example concerning diverse execution control models

as well as reconfiguration possibilities are also presented.

Finally, the last example (section 7.7), although not completely implemented, aims to highlight the interest on enabling flexible manipulation of Structural Patterns through Operators, as it is proposed in our model. Such is illustrated through possible reconfiguration scenarios for a simulation of a Database access application.

7.2 Configuring Distributed Systems Topologies

Due to the inherent complexity of distributed systems and applications, a topology is identified in [192] as a useful abstraction to simplify the understanding and construction of a distributed system's architecture. We discuss in this section the usage of our model on assisting the configuration of common distributed systems topologies.

Topologies can be identified at different levels, e.g. physical, logical, connection, or organisational, and can be considered in terms of the information flow [192]. Moreover, it is possible to identify in those levels a set of essential topologies, which are also the basis to build a group of more complex topologies that are common in distributed systems¹. Consequently, the possibility of reusing those typical basic topologies may, in our opinion, help configuring new distributed systems. Our goal with this section is simply to illustrate how the model proposed in this dissertation can be used to specify most of those typical topologies. Such is achieved by manipulating the basic Structural Patterns in the model through the proposed Structural Operators and combining the resulting structural configuration with Behavioural Patterns.

7.2.1 Basic Topologies

In [192], four elementary topologies are identified and discussed, namely *Centralised*, *Ring*, *Hierarchical*, and *Decentralised*.

A- Centralised

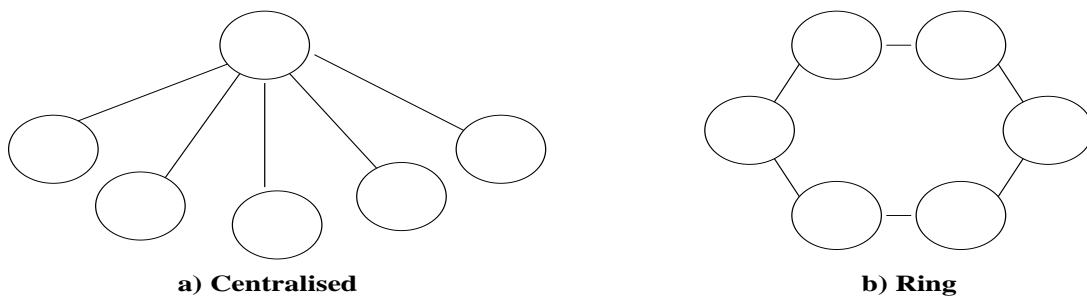


Figure 7.1: The *Centralised* and *Ring* distributed systems topologies.

Centralised systems, part a) of Figure 7.1, are the most familiar form of topology. Those are usually known to support the “Client/Server pattern” which is extensively used in distributed

¹Some criteria for evaluating topologies and discuss their relative merits in terms of the existing system designs are also presented in [193].

systems [193]. In this pattern, many clients connect to the server which centralises all function and information, making scalability one of its major problems, which only may be partially overcome by providing a fast server.

In our model, the Star Structural Pattern associated with the Client/Server Behavioural Pattern (where the “nucleus” of the star is the “server”) maps the centralised topology directly. Moreover, the number of satellites of the Star Structural Pattern may be easily changed through the *Increase/Decrease* Structural Operators. As discussed in section 5.2.2 the centralised topology is one example of a *Regular SB-PT* and, consequently, the insertion/deletion of elements was defined as not disrupting the overall *Client/Server* behaviour, and the new satellites are automatically annotated with a behavioural role, namely as “clients”.

B- Ring

The ring topology, part *b*) in Figure 7.1, is also described as a commonly found solution in distributed and parallel systems. For example, the ring topology frequently underpins a cluster of machines, globally providing a distributed service, in the context of a local network owned by a single organisation. Considering the present and planned capabilities of Grid environments, the support to ring topologies can be extended to a larger scale, the restriction of single organisation ownership can be overcome, and with the support of a reasonably fast connection between the Ring’s elements distributed on the network.

In our model, the Ring Structural Pattern, combined with Behavioural Patterns such as Stream Dataflow, Producer/Consumer, Client/Server, etc, can be used to specify the ring topology. If all stages in a Ring Structural Pattern are ruled by the same behavioural role(s) within a single Behavioural Pattern, the result is also a *Regular SB-PT*. This definition represents the possibility of changing both the structure and the behaviour of ring topology-based architectures in a transparent way through the Structural and Behavioural Operators.

C- Hierarchical

Hierarchical systems are common in distributed systems, namely in the Internet (e.g. the *Domain Name Service*), to enable access to distributed resources and to ease the dissemination of information [192]. The left-hand side of Figure 7.2 (part c) shows an example of a hierarchical topology, whereas its right-hand side displays how that hierarchy is configured using Structural Patterns and Operators. A hierarchy may be considered as a recursive definition of a tree, which can be supported in our model through the manipulation of the Star Structural pattern by Structural Operators. The construction of a hierarchical topology can be done step by step in a Problem Solving Environment, and can also be automated through a script.

In the right-hand side of Figure 7.2, a Star Structural Pattern Template (S-PT) composed of a nucleus and two satellites is processed by a script to build the presented example of a hierarchical topology. First of all, the template can be generated with the *Create(StarSP, “starPT”, 3)*, and it is subsequently replicated four times (step 1 in the script in Figure 7.2) resulting in five Star S-PTs: “starPT”, “star1PT”, ..., “star4PT”. It is assumed that the replica names are defined in the invocation of the *Replicate* operator and that they are unique pattern identifiers within the configuration.

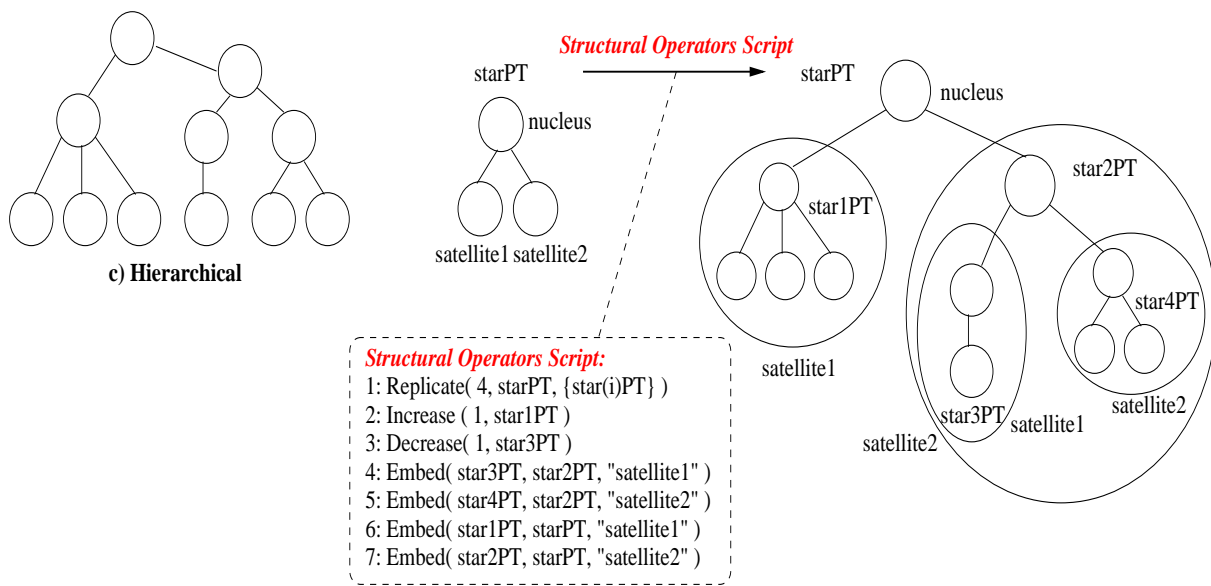


Figure 7.2: The Hierarchical distributed systems topology (c) and its modelling through the Star Structural Pattern Template manipulated by Structural Operators.

To form the left side of the hierarchic topology example, the “star1PT” has its number of satellites increased by one (step 2 in the script), and it is embedded in the “satellite1” of “starPT” through the *Embed* Structural Operator (step 6). The right-hand side of the final hierarchy is built by: a) decreasing the number of satellites of “star3PT” by one (step 3 in the script); b) embedding it in the “satellite1” of the “star2PT” (step 4); c) embedding “star4PT” in “satellite2” of the “star2PT” (step 5); and d) embedding the “star2PT” in the “satellite2” of the “starPT”. The nucleus of “starPT” is the root of the hierarchy. Please note that *Compound Structural Operators* might be used to build the hierarchy:

```
Embed(star4PT,Embed(star3PT,star2PT,'satellite1'),'satellite2')
Embed(star2PT, Embed(star1PT,starPT,'satellite1'), 'satellite2')
```

Through similar scripts, our model allows the automated construction of hierarchies and, additionally, the instances of the basic Structural Pattern used (i.e. the Star) are still accessible and manipulable. For example, the application of the *Increase* operator allows the addition of new elements to the embedded “star1PT”, “star3PT”,etc, as well to the outer pattern “starPT”.

As for the typical information flow in the hierarchy (as described in [192]), which starts at the root down to the leaves, it may be represented by applying the Streaming Behavioural Pattern to the nucleus of each of the Star patterns in the built star-based configuration. However, other Behavioural Patterns may be still (directly) applied to the inner Star S-PTs instead.

D- Decentralised

Figure 7.3 shows an example of a decentralised topology, which is typical of peer-to-peer systems. None of the Topological Structural Patterns maps directly to this topology. Nevertheless, by having our model supported by a Problem Solving Environment like Triana, as described in Chapter 6, the user may therefore connect the components directly and build the decen-

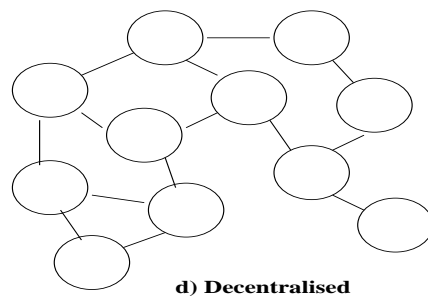


Figure 7.3: *The Decentralised topology.*

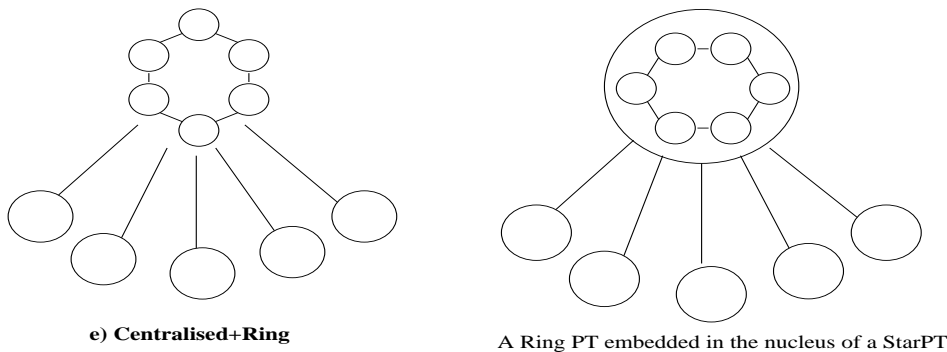
tralised topology. This configuration can be included in the Triana repository for subsequent use, refinement, and instantiation.

Although not discussed in this work, it would be desirable to define the decentralised topology as a basic Structural Pattern of its own, and to allow the application of Structural and Behavioural Operators to its templates. For example, the behaviour could be defined on a per-element basis through the *DefineRoleBehavPatt*(*P*, *B-P*, {*element*, *role*}) Behavioural Operator, or a unique Behavioural Pattern could be applied to all elements in the topology forming a regular pattern. This would be the case of the *Peer-to-Peer Behavioural Pattern* where new elements in the configuration would have a similar role (i.e. “peer”) to the pre-existing elements. Structural Operators such as *Replicate*, *Replace*, *Group*, or even *Embed* would, at first, be straightforward, but other operators such as *Increase/Decrease* would require a deeper study.

7.2.2 Hybrid Topologies

Based on the basic topologies described in the previous section, this section discusses a set of hybrid topologies formed from those basic ones, as presented in [192].

E- Centralised+Ring



A Ring PT embedded in the nucleus of a StarPT

Figure 7.4: *The hybrid Centralised+Ring topology, and its configuration by embedding a Ring Pattern Template into the nucleus of a Star Pattern Template.*

The left-hand side of Figure 7.4 presents an example of an hybrid topology that results from the combination of the ring and centralised topologies. This can describe, for instance,

the configuration of a service supported by a ring of servers for load balancing and fail-over, and the system as a whole is seen as a centralised system from the clients' point of view.

In our model, the representation of such hybrid topology is supported by a Ring S-PT embedded in the nucleus of a Star S-PT (as shown in the right-hand side of figure 7.4). Moreover, the *Client/Server* Behavioural Pattern when applied to the Star S-PT may represent the interactions between the clients and the service as whole. In turn, the servers (i.e. the elements in the embedded Ring S-PT) may interact between them according to different Behavioural Patterns (e.g. *Streaming*, *Itinerary*, etc). Please note that both the embedded pattern and the encloser pattern may be defined as *Regular SB-PTs*, and are also directly manipulable through Operators. For example, the addition of an extra element to the ring in the topology (e.g. to accommodate another server) can be supported by the *Increase* operator applied to the embedded *Pattern Template (SB-PT)*, and the new element is annotated with the same behaviour as the other elements in the ring.

F- Centralised+Centralised

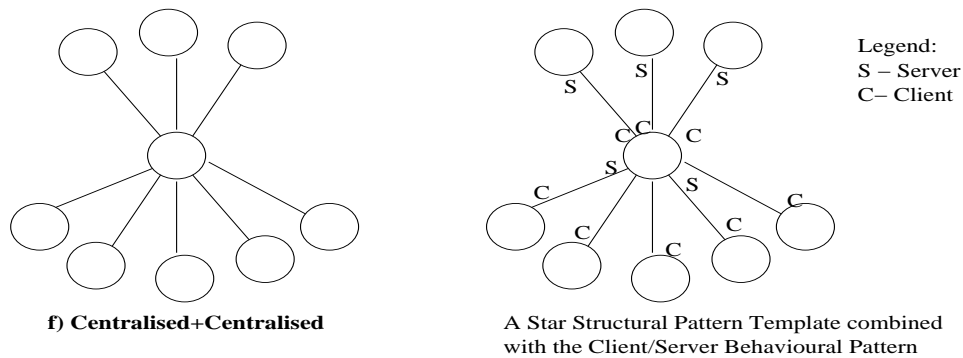
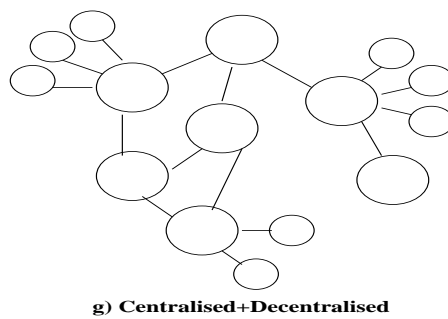


Figure 7.5: The hybrid *Centralised+Centralised* topology, and its configuration by the combination of the *Star Structural Pattern* and the *Client/Server Behavioural Pattern*.

Another hybrid topology represents a centralised system that is itself a client of one or more other servers. The left-hand side of Figure 7.5 represents such a “Centralised+Centralised” topology. The right-hand side of the Figure shows how this may be described, namely, by combining the *Star Structural Pattern* with the *Client/Server Behavioural Pattern*. The nucleus of the Star represents the server that handles requests from the clients, and it is, itself, a client of another set of servers. In this case, the Star S-PT in the Figure is combined with the same Behavioural Pattern but it is not regular, since the roles of all elements is not uniform: the nucleus is both a client and a server, and the satellites may either clients or servers disallowing the possibility of pre-defining a role for the new satellites. As such, the behaviour has to be defined in a per-element basis through the *DefineRoleBehavPatt(P, B-P, {element, role})* Behavioural Operator.

G- Centralised+Decentralised

The *Centralised+Decentralised* topology (left hand-side of Figure 7.6) represents a common situation in current peer-to-peer systems where some peers have a centralised relationship with a



g) Centralised+Decentralised

Figure 7.6: The hybrid Centralised+Decentralised topology.

“supernode” which in turn interact with other peers, in a decentralised way, to respond to requests. Our topological Structural Patterns do not support this topology directly. Nevertheless, once the *decentralised topology* described before is made available as a pattern template, the user may then embed the necessary Star S-PTs into the nodes of that topology which then represent the centralised access to that particular node.

7.3 Configuring a Problem Solving Environment

In this section, a typical *Problem Solving Environment (PSE)* configuration example is provided to describe activities that are commonly required to manage an application. This example illustrates some of the applicabilities of patterns and operators on modelling similar environments, and it was first discussed in [44].

7.3.1 A Typical PSE Example

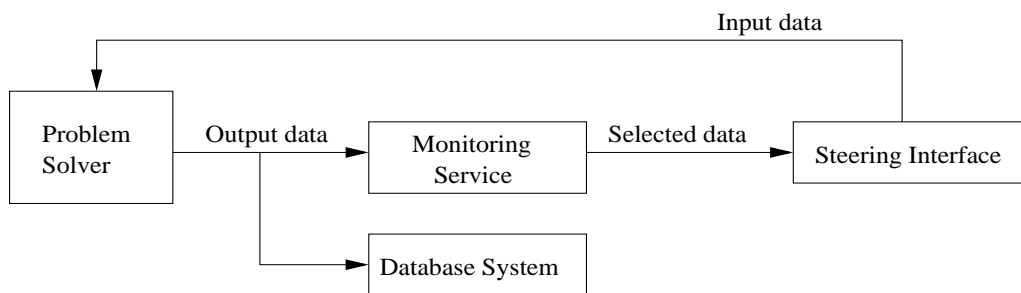


Figure 7.7: A PSE supporting the active steering of a Problem Solver. The arrows represent the flow of data.

Figure 7.7 presents an example of the structure of a PSE combining different types of services, which frequently appear in applications. The *Problem Solver* component represents, for example, a service running some scientific experiment that continuously produces data. An instance of such a service may be, for instance, a wave generator or a matrix solver. After receiving some initial input parameters, the service starts producing data that can be analysed at run-time or stored for “post-mortem” analysis. The *Problem Solver* service may be steerable, meaning that its input parameters can be changed while the service is executing. By

adjusting the input parameters a user may, for example, generate and then visualise particular behaviours using this service.

Steering is frequently supported by two types of services: a *Monitoring Service* and a *Steering Interface*. The *Monitoring Service* is used to register relevant output data or events produced by the Problem Solver. The data/events are filtered by the *Monitoring Service* and they are passed to a *Steering Interface* that shows them to the user in a pre-defined format. Consequently, a user may use the *Steering Interface* to undertake “what if” scenarios – generally by defining new values for the *Problem Solver*’s input data. Furthermore, one may consider that several users have access to the *Steering Interface*, thus requiring some coordination over changing the parameters of the *Problem Solver*.

This kind of applications may also include another service, namely a *Database System* (as represented in Figure 7.7) to store all the output produced by the *Problem Solver*. This may enable a user to reconfigure the PSE without requiring the *Problem Solver* to be stopped, i.e. all data is saved in the *Database System*. Moreover, a user may re-examine output data for additional processing after the *Problem Solver* terminates its execution (based on pre-defined behaviour or as a result of a fault). These scenarios are illustrated, respectively, in Figures 7.8 and 7.9, and in Figure 7.10.

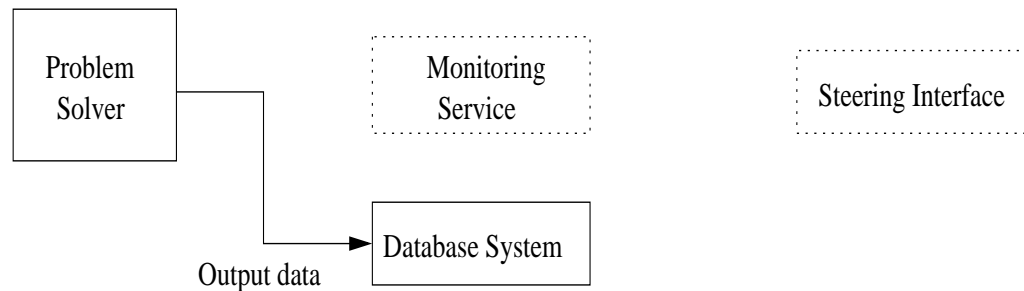


Figure 7.8: The *Monitoring service* is stopped and consequently the *Steering interface* also stops. The output data is not lost because it is being saved in the *Database system*.

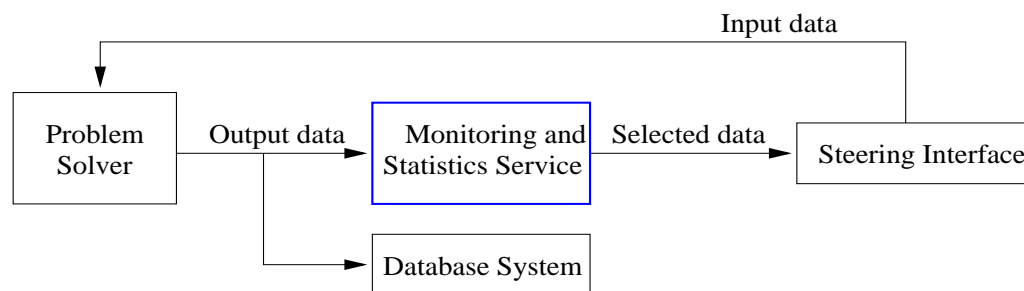


Figure 7.9: The initial *Monitoring service* is replaced with a more complex one (*Monitoring and Statistics service*), which is activated to continue the filtering of the output data.

In Figure 7.8, the *Monitoring Service* is stopped so that it can be replaced with a more complex tool like the *Monitoring and Statistics service* in Figure 7.9; in the meantime, the *Problem Solver* continues its execution and its output is saved in the *Database System*.

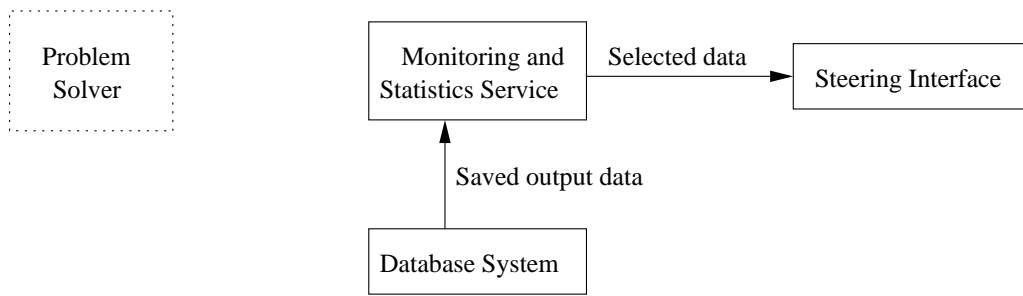


Figure 7.10: After the *Problem Solver* terminates its execution, data can be re-analysed.

The alternative scenario is illustrated in Figure 7.10: after the *Problem Solver* terminates its execution, its output can be processed, either from the beginning or from the point at which the *Monitoring Service* was being replaced (and that would otherwise be lost). In this case, the *Database System* acts as a temporary buffer. Clearly, in this alternative scenario depicted in Figure 7.10, the *Steering Interface* cannot be used anymore to parameterise the application.

The next two sections identify which Structural and Behavioural Patterns could be used to configure the PSE outlined in Figure 7.7. Subsequently, section 7.3.4 describes the application of Structural Operators to build that PSE, and section 7.3.5 discusses the application of Behavioural Operators to control the PSE's execution. The initial modelling of the example follows the *Basic Methodology* as previously described in section 3.1.3. Subsequently, the modification of that configuration is based on the concepts discussed in 5.2 and 5.3.

7.3.2 Structural Patterns in Use

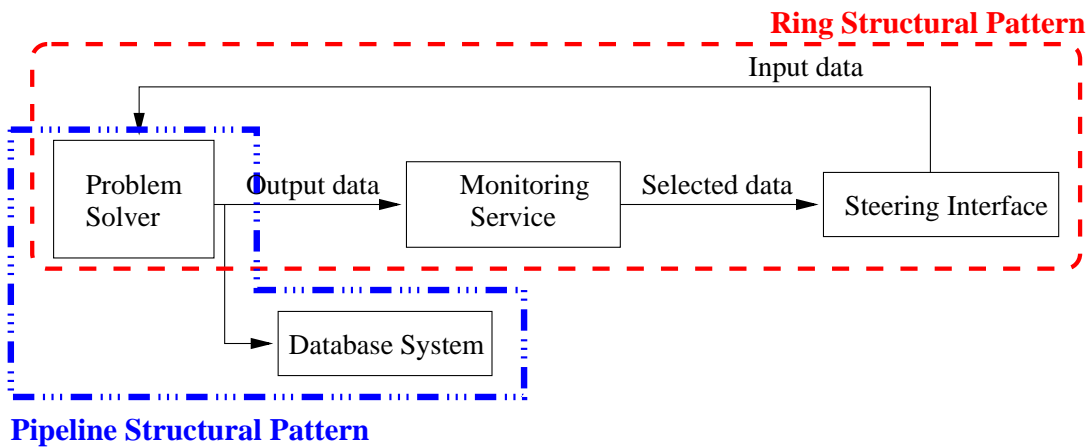


Figure 7.11: Identification of the Ring and Pipeline patterns in the PSE example.

The analysis of the interactions between the components in 7.7 suggests two main Structural Patterns which are represented in Figure 7.11: a) a *Ring Structural Pattern* may configure the interactions between the *Problem Solver*, the *Monitoring Service* and the *Steering Interface*; and b) a *Pipeline Structural Pattern* may connect the *Problem Solver* to the *Database System*.

To represent such configuration the user would define a Ring Structural Template (S-PT) with three elements, and a Pipeline Structural Template with two elements. One way to com-

bine the two patterns is by embedding the Pipeline S-PT into one of the elements of the Ring S-PT forming a *Hierarchical Pattern Template*. Structural Operators provide this kind of pattern combinations, as will be described in one of the following sections.

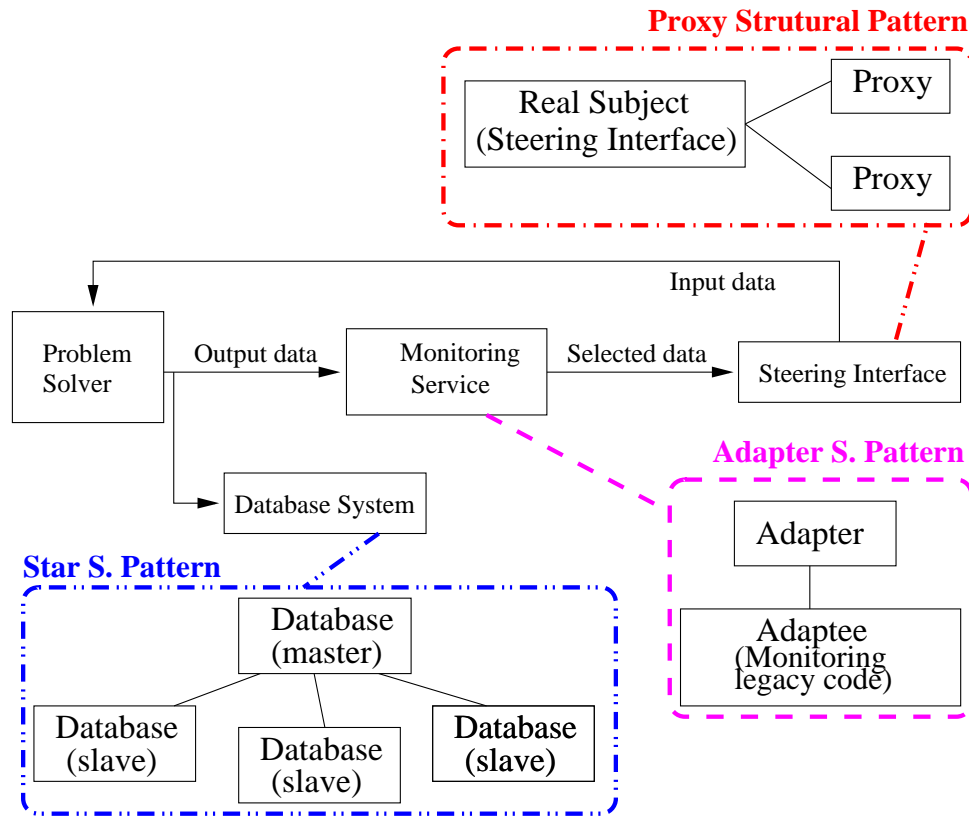


Figure 7.12: Identification of the Star, Adapter, and Proxy patterns in the PSE example.

Figure 7.12 identifies three more Structural Patterns, namely for configuring the individual services. For example, the *Star Structural Pattern* may represent the *Database System* supposing that the system is composed of a set of distributed Database sub-systems. These sub-systems are the *Satellites* in the Ring's structure, and they are controlled by a *Master* Database system acting as a coordinator. The Figure shows that Star with three satellites.

The second example in Figure 7.12 presents the possibility of using the *Adapter Structural Pattern* for the Monitoring service. This service may be supported by legacy code which needs to be adapted in order to interact with the other services. This Adapter pattern would be embedded in the second element of the Ring.

Finally, one way to represent the possibility of sharing the *Steering Interface* among multiple users is through the *Proxy Structural Pattern*. Each user has a Proxy to access the central service which in turn controls the concurrent accesses. Therefore, the *Steering Interface* placed at the *subject* element of the Proxy pattern coordinates the concurrent requests from users to tune the *Problem Solver*. The Figure shows a Proxy S-PT with two proxies for two users.

7.3.3 Behavioural Patterns in Use

Taking as a basis the Structural Patterns illustrated in Figures 7.11 and 7.12 this section enumerates some applicable Behavioural Patterns. Please see table 3.1 in section 3.3.7 for a more

complete list.

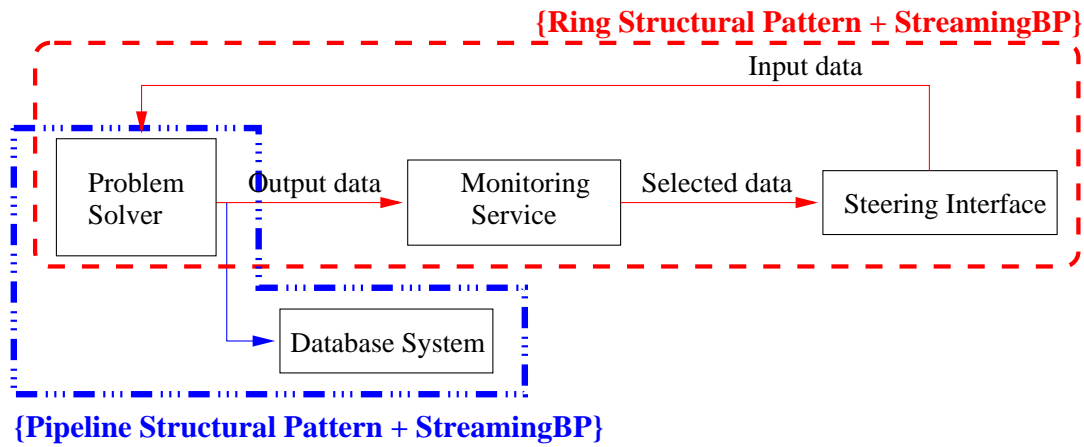


Figure 7.13: Combination of the Ring pattern with the Producer/Consumer Behavioural Pattern, and the Pipeline pattern with the Streaming Behavioural Pattern.

The following combinations of Structural and Behavioural patterns are depicted in Figure 7.13 (the first three enumerated cases) and in Figure 7.14 (the last three enumerated cases):

1. The *Streaming* pattern may represent the interaction between the Problem Solver and the Monitoring Service. However, if the Monitoring service only requires a sub-set of the data produced by the Problem Solver, then such interaction might be represented by the *Observer* pattern.
2. The *Streaming* pattern may also be used to represent the control and data flows between:
a) the Monitoring service (source of the selected data) and the Steering Interface (destination of the selected data) in the Ring pattern (Figure 7.11); b) the Steering Interface (source of the input data to tune the application) and the Problem Solver (destination of the input data).
3. The *Streaming* pattern is once again used to define the data and control dependencies for the elements in the Pipeline Structural Pattern. This Structural Pattern connects the Problem Solver and the Database System, and the *Streaming* behaviour represents the continuous flow of data generated by the Problem Solver and that needs to be maintained in the Database System.
4. The *Master/Slave* pattern can represent the behaviour of the Database System (Figure 7.12): a master controls and distributes requests to the slaves.
5. The *Client/Server* pattern can represent the interaction between the Steering Interface (server) and its proxies (clients) that redirect users' requests to access the Steering service.
6. The *Adapter* Structural Pattern that gives access to the legacy code to support the Monitoring service can be combined with the *Service Adapter* Behavioural Pattern which "attaches additional properties or behaviours to an existing application to enable it to be invoked as a service" [34].

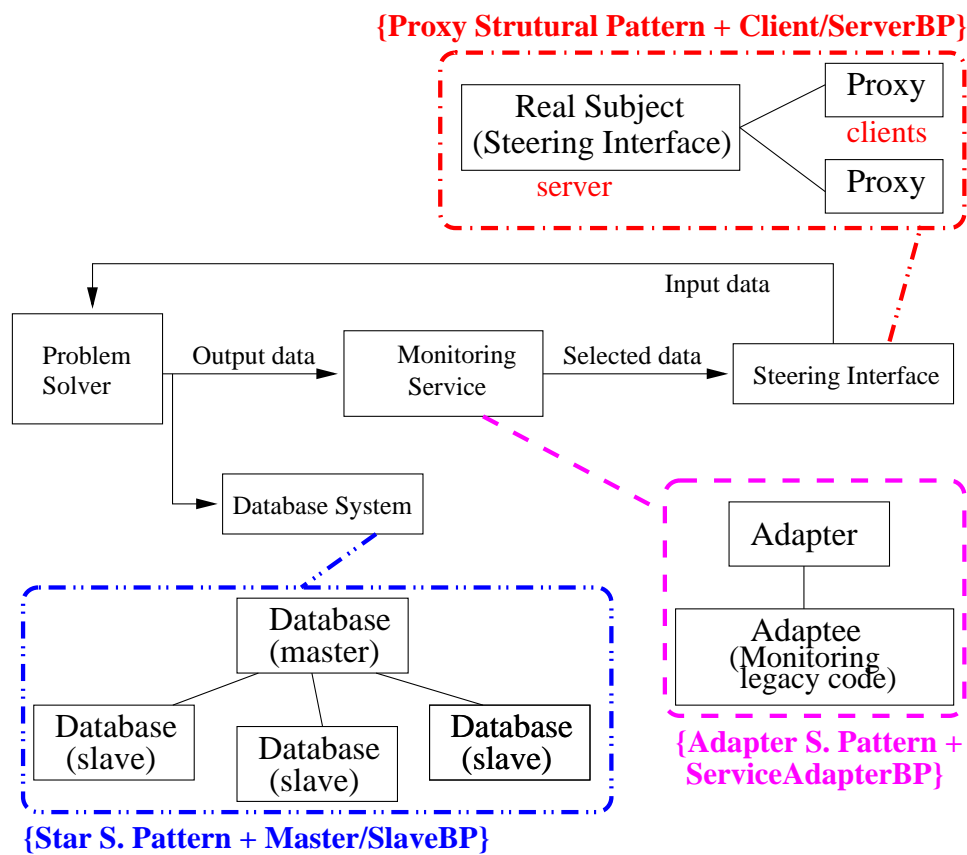


Figure 7.14: Combination of: the Star SP with the Master/Slave Behavioural Pattern; the Adapter SP and the Service Adapter Behavioural Pattern; and the Proxy SP with the Client/Server Behavioural Pattern.

Having identified the Structural and Behavioural patterns, the following sub-section describe the usage of Structural Operators in order to build the desired configuration. The subsequent sections describe the execution control of the final application through Behavioural Operators, as well as some reconfiguration scenarios.

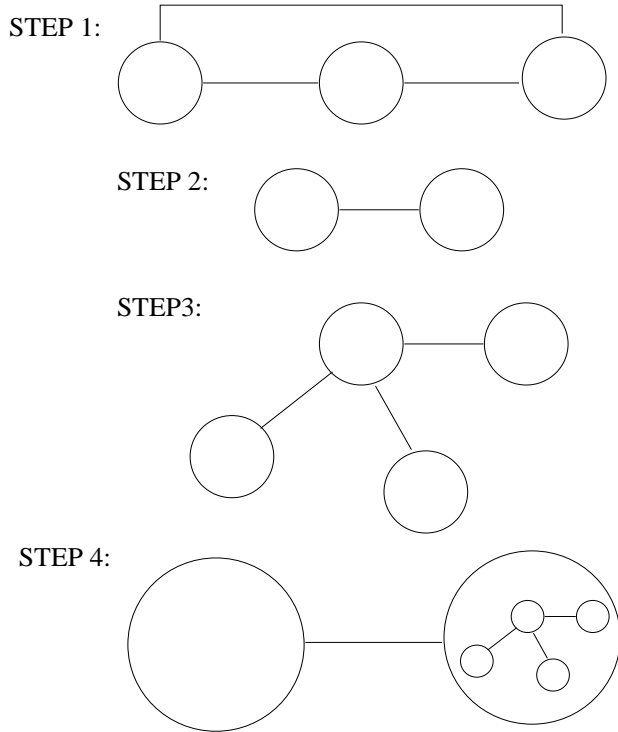
7.3.4 Structural Operations

Figures 7.15 and 7.16 describe a possible sequence of steps to build the PSE configuration shown in Figure 7.7, according to the patterns identified in Figures 7.11 and 7.12. The correspondent Structural Operator sequence is divided in parts along with the explanation of a sub-set of the steps, as presented ahead. Lines in these sequences are numbered according to the enumerated steps.

The following operator sequence steps are presented in Figure 7.15:

```
1: Create( RingSP, ``PSE'', 3 )
2: Create( PipelineSP, ``DataStoring'', 2 )
3: Create( StarSP, ``DatabaseSystem'', 4 )
4: Embed( DatabaseSystem, DataStoring, ``cph2'' )
```

Step 1 The user creates a Ring Structural Pattern Template (S-PT) with three component place



Step 1– Creation of a ring PT with three component place holders (for the problem solver, the monitoring service, and the steering interface).

Step 2 – Creation of a pipeline PT with two component place holders (for the problem solver and the database system).

Step 3 – Creation of a star PT for the database system (the front–end will be the nucleus and the slaves will be the satellites).

Step 4 – Embedding of the star PT into the pipeline PT built in step 2.

Figure 7.15: Initial steps for building the PSE depicted in Figure 7.7.

holders (CPHs) to represent the components connecting the Problem Solver, the Monitoring Service, and the Steering Interface.

Step 2 Next, the user creates a Pipeline S-PT named “DataStoring” with two CPHs to represent the connection between the Problem Solver and the Database System. This pipeline will be embedded in the first component place holder of the ring, but first the user creates a S-PT to represent the Database System.

Step 3 The user creates a Star S-PT named “DatabaseSystem” with three satellites that will be instantiated to the Database sub-systems.

Step 4 The user applies the *Embed* Structural Operator over the Pipeline S-PT with the Star S-PT to be embedded in the second component place holder (“cph2”) of the Pipeline S-PT.

The following operator sequence steps are presented in Figure 7.16:

```
5: Embed( DataStoring, PSE, ``cph1`` )
6: Create( AdapterSP, ``MonitoringSv`` )
7: Create( ProxySP, ``SteeringInt`` )
8: Increase( 1, SteeringInt )
```

Step 5 The user applies the *Embed* Structural Operator to include the Pipeline S-PT obtained in step 4 into the first component place holder (“cph1”) of the Ring S-PT (previously defined in step 1).

Step 6 Next, the user creates an Adapter S-PT named “MonitoringSv” to represent the Monitoring service.

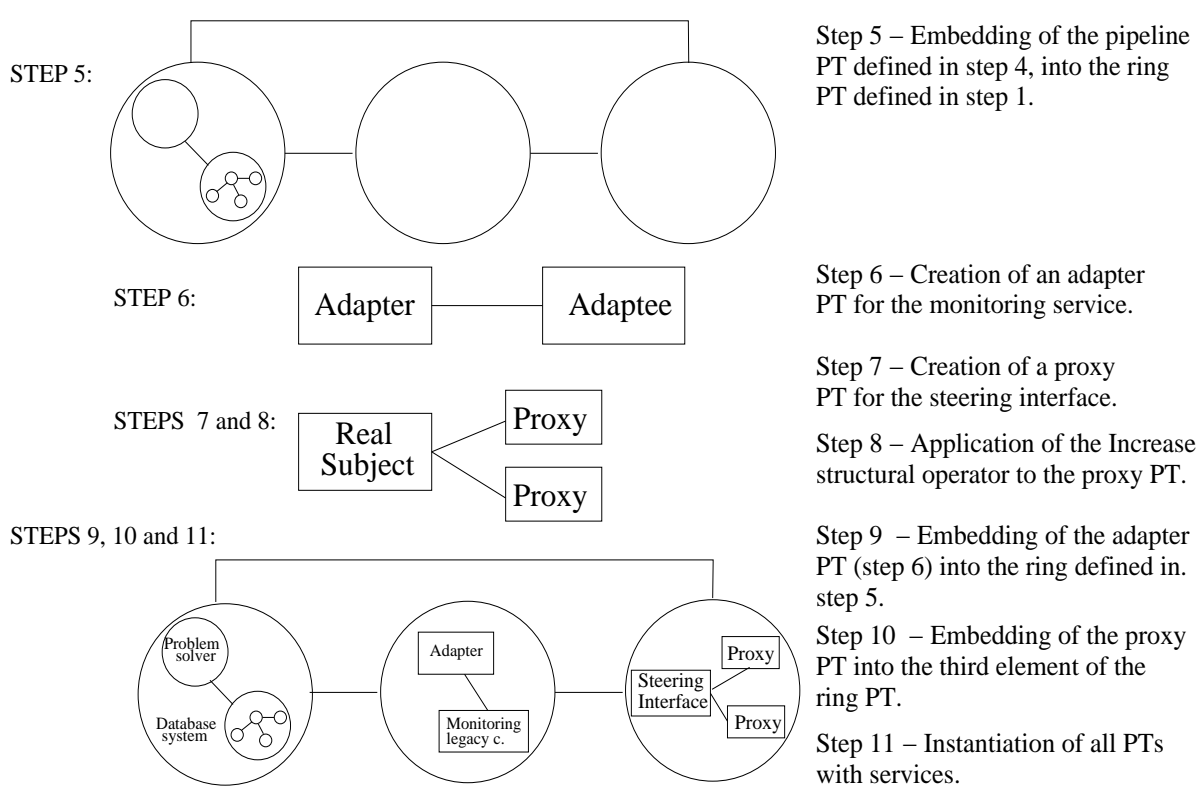


Figure 7.16: Final steps for building the PSE depicted in Figure 7.7.

Steps 7 and 8 The user creates the structure for the Steering Interface (which will be accessed by other users). To achieve this, the user creates a Proxy S-PT named “SteeringInt” and then its proxy elements are increased by one through the application of the *Increase Structural Operator*.

The following operator sequence steps are also presented in Figure 7.16:

```
9: Embed( MonitoringSv, PSE, ``cph2`` )
10: Embed( SteeringInt, PSE, ``cph3`` )
```

Steps 9 and 10 The user embeds the Adapter S-PT and the Proxy S-PT in the ring’s second and third component place holders, respectively (i.e. “cph2” and “cph3”).

Step 11 Finally, the user instantiates all pattern templates with the selected services. This step requires the application of the *Instantiate(P, position, component)* operator (described in section 5.2.3) in case the configuration is generated by a script. Alternatively, the instantiation of the elements may be done through a GUI of a workflow tool, like the one provided by the implementation of patterns/operators over the *Triana* Problem Solving Environment, as described in section 6.4.

Having defined the structural configuration, the user may now apply the appropriate Behavioural Patterns, as defined in section 7.3.3, and run the application using the Behavioural Operators to control its execution. Such is described in the next section.

7.3.5 Behavioural Operators in Use

A *Behavioural Operator sequence* following the Structural Operator sequence defined in the previous section may be used to define the behaviours that rule each of the Structural Patterns in the current example according to what was presented in Figures 7.13 and 7.14. The final configuration is depicted in Figure 7.17. In all cases, we define that the result is a *Regular Pattern Instance (PI)* as defined in section 5.2.4, meaning that a single Behavioural Pattern is applied to one individual Structural Pattern, and the behavioural role of future elements within each Structural Pattern is pre-defined.

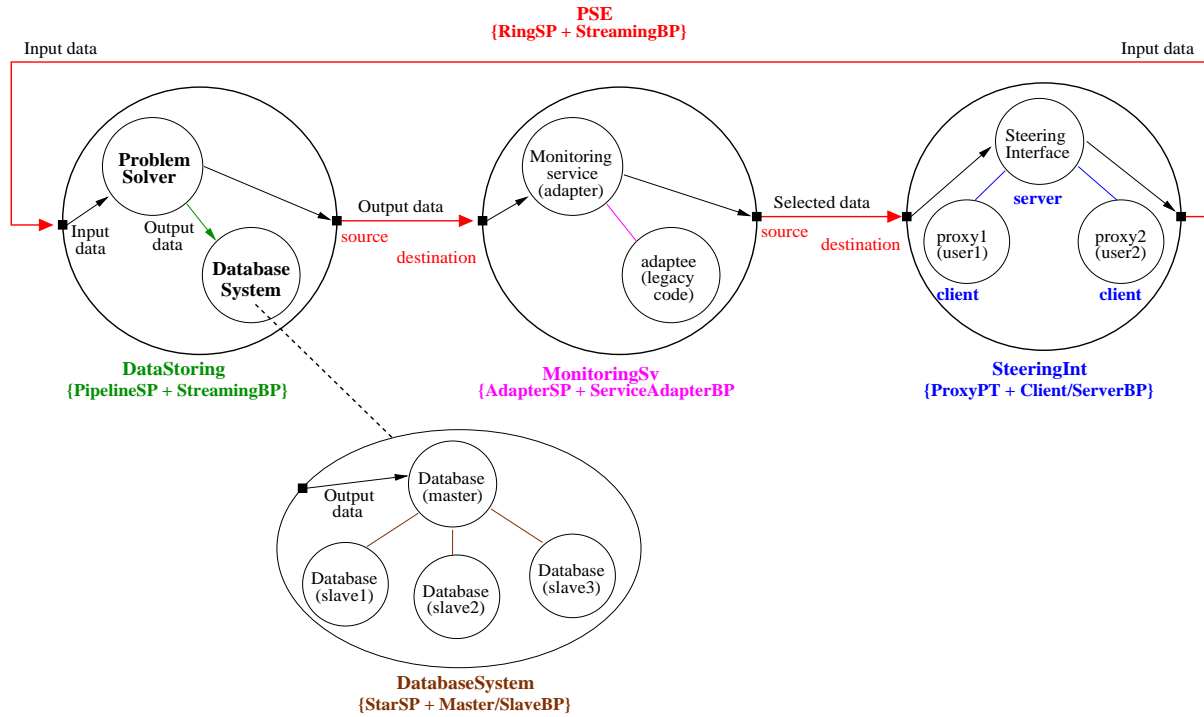


Figure 7.17: The final configuration for the PSE example.

Operator sequence steps generating the final PSE configuration illustrated in Figure 7.17:

```
12: DefineBehavPatt( PSE.DataStoring.DatabaseSystem, ``Master/Slave`` )
13: DefineBehavPatt( PSE.DataStoring, ``Streaming`` )
14: DefineBehavPatt( PSE, ``Streaming`` )
```

Step 12 The embedded “DatabaseSystem” pattern is composed with the *Master/Slave* Behavioural Pattern where the “nucleus” of the Star pattern is defined as the *Database master*. The tool/service that instantiates the nucleus is to select to which satellite, i.e. “Database slave”, data will be sent to be stored. The “DatabaseSystem” represents a distributed (federated) database system that, for example, is common in distributed and Grid environments. Please note that this operation is possible since the embedded “DatabaseSystem” pattern is still directly accessible for manipulation. As defined in the operator invocation, this embedded pattern is accessed through the concatenation of the patterns’ identifiers starting from the outer pattern, i.e. “PSE”. Specifically the “DatabaseSystem” is referenced through the identifier: “PSE.DataStoring.DatabaseSystem”.

Step 13 The flow of data generated by the “Problem Solver” in the “DataStoring” Pipeline pattern is ruled according to the *Streaming Behavioural Pattern*, where the destination of the data is the *Database master*. This means that the embedding operation previously defined in *step 4*, i.e. *Embed(DatabaseSystem, DataStoring, “cph2”)*, will imply a structural connection representing the data flow to the nucleus of the star, i.e. *Database master*.

Step 14 The *Streaming Behavioural Pattern* is to coordinate at run-time all elements in the ring-based “PSE” pattern. The “Problem Solver” is defined as a “source” of data and the “MonitoringSv” is defined as the “destination” of that generated data.

Furthermore, the “MonitoringSv” is also defined as a “source”, i.e. filtered data at the Monitoring Service is sent to the next stage in the Ring, namely the Steering Interface (“SteeringInt”). This means that the “adapter” element within that Adapter-based pattern that represents the Monitoring Service consumes data generated by the “Problem Solver” and, additionally, data filtered at the “adaptee” element (i.e. the legacy code that implements the monitoring) is also sent by the “adapter” to be consumed by the “SteeringInt” pattern. In this element, the “destination” of the selected data is the “Steering Interface” that instantiates the “subject” element within the Proxy-based “SteeringInt” pattern.

Finally, the data for application tuning generated by the Steering Interface is considered also to be consumed by the “Problem Solver”. Therefore, the “SteeringInt” as well as the “Problem Solver” are alike considered as “sources” and “destinations” of data.

```
15: DefineBehavPatt( PSE.MonitoringSv, ``Service Adapter`` )
```

```
16: DefineBehavPatt( PSE.SteeringInt, ``Client/Server`` )
```

Step 15 The *Service Adapter Behavioural Pattern* is combined with the Adapter pattern (named “MonitoringSv”) to provide access to the legacy code as a service. Please note that we assume that the data generated at the *Problem Solver* is delivered to the adapter element (“Monitoring service” in Figure 7.17).

Step 16 Finally, the Proxy pattern that represents the Steering Interface is combined with the *Client/Server Behavioural Pattern* where the proxy elements in the pattern are the “clients”. These proxies forward requests from the users in order to access the Steering Interface (i.e. the “subject” element within the Proxy pattern and that behaves as the “server”).

After the above Behavioural Operator sequence (steps 12 to 16), the final configuration represented in Figure 7.17 is ready to be executed. This *PSE Pattern Instance (PI)* can be manipulated through *Execution operators* in different ways. For instance:

1. The application of the *Start(PSE)* operator launches the execution of all components/services. Through the proxies that give access to the *Steering Interface*, two (remote) users may tune the *Problem Solver* according to data collected by the *Monitoring Service*.
2. The application of the *Start(PSE)* and *Limit(time_interval, PSE)* in sequence allows limiting the time the PSE PI is allowed to execute. Upon expiration of the time defined in “time_interval”, the execution of all elements is terminated.

3. The application of the *Restart(time_interval, PSE)* provides the periodic execution of the complete application. For example, the PSE may be restarted every day at the same time, and when this automatic re-execution is no longer desired, the *TerminateRestart(PSE)* operator discontinues that periodic execution.

Please note that in these three examples the *Start*, *Limit* and *Restart/TerminateRestart* operators are assumed to act recursively upon the hierarchy forming the *PSE* Pattern Instance.

Moreover, it is also assumed that the coordination between the defined Behavioural Pattern for the *PSE* PI and the behaviours of the embedded PIs is implementation dependent. For example, whereas the overall enclosing pattern, i.e. the *PSE* PI, is ruled by the *Streaming* Behavioural Pattern, the PI embedded at the second stage of that Ring-based pattern, namely the “Monitoring service”, is ruled by the *Service Adapter* Behavioural Pattern. Therefore, it is necessary to coordinate the reception of data at that second stage with the invocation of the embedded “Monitoring service” that will process that received data. Implementation-wise, such coordination is to be guaranteed by the *pattern controller* (defined in section 6.2.1) of each embedded pattern in orchestration with the *pattern controller* of the enclosing pattern.

We recall that the detailed study of the behavioural coordination in Hierarchical patterns, although fundamental, is out of the scope of this thesis, and it is deferred to future work.

The next sub-section presents a few reconfiguration scenarios of the *PSE* example, which include the ones discussed in section 7.3.

7.3.6 Reconfiguration Scenarios

Starting from the configuration defined in the previous section (Figure 7.17) it is possible to define a few reconfiguration scenarios resulting from pattern manipulation through Structural and Behavioural Operators. Some scenarios have to be accomplished at development time, whereas others may be done at execution time.

For both cases, we may assume that the application reconfiguration is restricted to a particular set of users. To that purpose, the *DefineOwners* Ownership Operator introduced in section 3.3.4 may be used to specify those access restrictions. For example, with the *DefineOwners(PSE, “PSEmanager”, “scientist1”, “scientist2”)* operation, it is possible to designate that only the user identified as the manager of the *PSE* example, and two scientists may operate the application. We recall again that the implications of the access restrictions associated to the *DefineOwners* operation are implementation dependent.

Replacing the Monitoring Service

Figures 7.8 and 7.9 in section 7.3.1 described a reconfiguration scenario where the *Monitoring service* is replaced with a more complex service, namely the *Monitoring and Statistics Service*, while the *Problem Solver* continues its execution. The hierarchical configuration of the “*PSE*” Pattern Instance defined for the *PSE* example permits its reconfiguration at run-time according to what was discussed in section 5.3. Specifically, the execution of part of the application representing the *PSE* example is suspended, while the rest of the components/services in the application remain in operation. In fact, only the execution of the service to be replaced, i.e. the

Monitoring service, is terminated while the *Problem Solver*, the *Database system*, and the *Steering Interface* are uninterrupted.

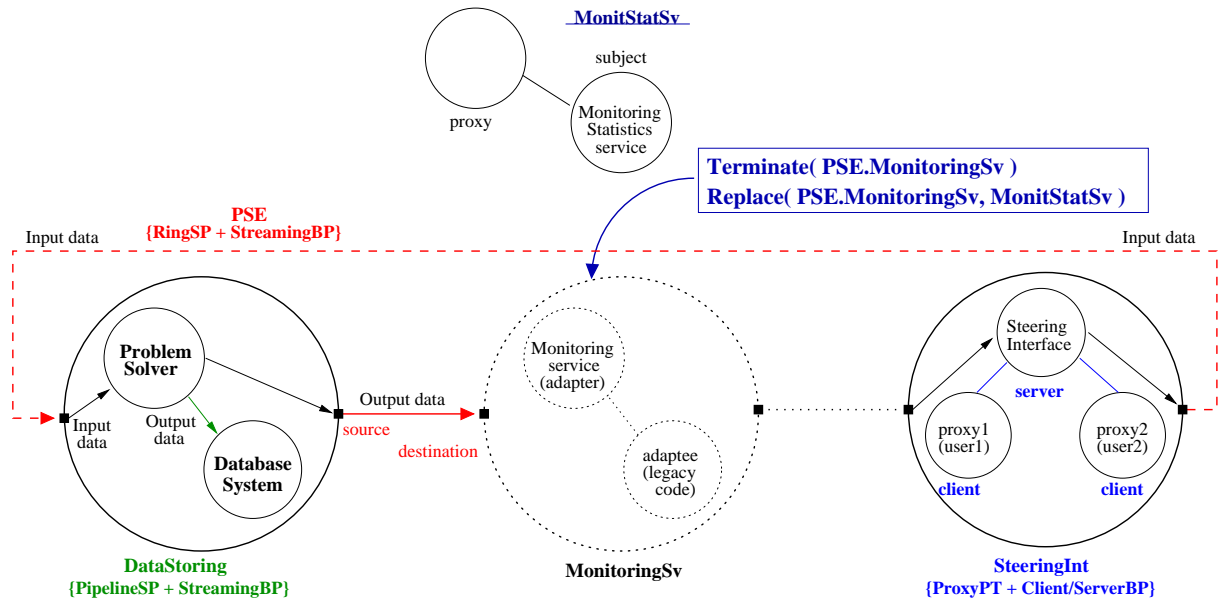


Figure 7.18: Applying the *Terminate Behavioural Operator* to replace the embedded “*MonitoringSv*” Pattern Instance (PI) for the “*Mon itStatSv*” pattern.

One way to achieve such reconfiguration through pattern operators may be:

1. To apply the *Terminate* operator to the “*MonitoringSv*” Pattern Instance (PI) located at the second stage of the Ring pattern. Consequently, the execution of the “*MonitoringSv*” pattern is aborted and data generated in the mean time by the *Problem Solver* is no longer processed at that second stage. Thus data is also no longer received by the “*SteeringInt*” PI located at the following stage. This is depicted in Figure 7.18.

Since the execution of the overall “*PSE*” PI was not aborted nor was terminated, the “*DataStoring*” PI continues its execution and data generated meanwhile by the *Problem Solver* is still saved at the *Database system*. Similarly, the “*SteeringInt*” PI remains active, which means that although data is not received from the previous stage and therefore it cannot be displayed at the *Steering Interface*, the users accessing its proxies may still tune the *Problem Solver*.

2. To apply the *Replace(PSE.MonitoringSv, MonitStatSv)* Structural operator so that the “*MonitStatSv*” pattern takes the place of the “*MonitoringSv*” pattern within the “*PSE*” PI. This pattern, also depicted in Figure 7.18, represents a Proxy-based PI which gives access to a (remote) *Monitoring and Statistics Service* that provides extra capabilities for processing data generated by the *Problem Solver*.
3. To launch the execution of the new pattern through the *Start(MonitStatSv)* Behavioural Operator. After the “*MonitStatSv*” starts executing, as depicted in Figure 7.19, the flow of data in the ring-based “*PSE*” PI continues normally.

Please note that the former “*MonitoringSv*” PI may either be added to a pattern repository (associated to an implementation support environment, like Triana) for later reuse, or may be deleted through the *Eliminate(MonitoringSv)* operator.

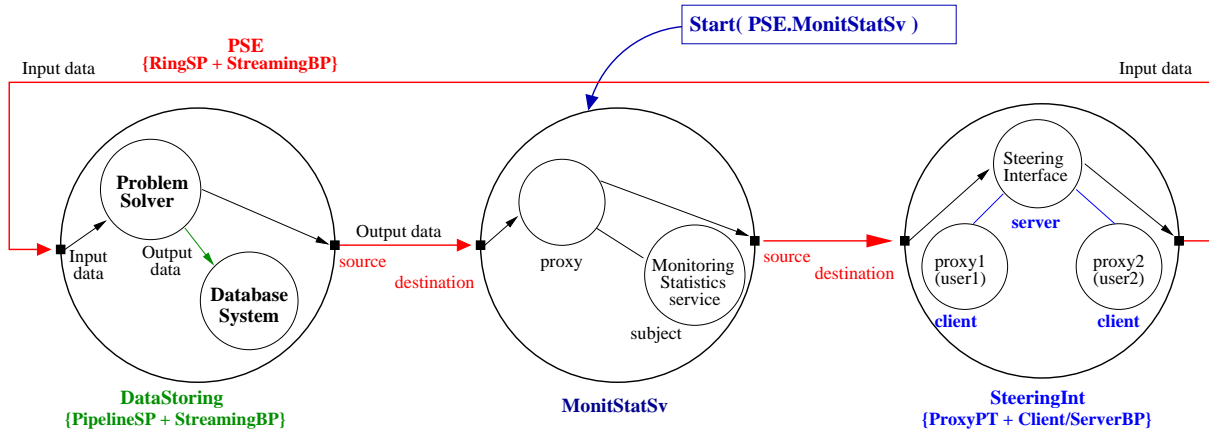


Figure 7.19: Applying the Start Behavioural Operator to the “Mon itStatSv” pattern to continue the execution.

Modifying the Steering Interface

As an example of an additional run-time reconfiguration of the “PSE” Pattern Instance, one may consider that the *Problem Solver* runs for a considerable amount of time and, meanwhile, the type of users that may access *Steering Interface* may change. For instance, the new users accessing that interface are passive (e.g. students observing the functionality of the steering interface), and therefore it may be necessary to guarantee that they are not allowed to change the *Problem Solver*’s parameters.

One possible way to accomplish such restriction is to change the Behavioural Pattern that coordinates the “SteeringInt” Pattern Instance (PI) as a whole, namely to the *Producer/Consumer* Behavioural Pattern (BP). The *Steering Interface* service (the “producer”) defined as the “subject” element at the Proxy-based PI sends data to the “proxy” components to be consumed. Users accessing these proxies no longer can submit requests to the *Steering Interface* and simply observe data generated by it.

An additional dynamic reconfiguration may be the possibility to change the number of passive users, e.g. incrementing the number of students, with the guarantee that new proxies for new users exhibit a similar behaviour to the existing ones. Concretely, the new proxies perform as “consumers” according to the ruling *Producer/Consumer* BP. We may assume that the possibility to accomplish this second reconfiguration is delegated by one of the owners of the PSE PI to a special user named “Steering Controller” (for example a teacher on a e-learning class).

One way to achieve such dynamic reconfigurations through pattern operators may be:

1. The execution of the *Steering Interface* is stopped as a result of the *Stop(PSE.SteeringInt)* Execution Operator. This is presented in Figure 7.20. Consequently, the execution of all components within the “PSE.SteeringInt” PI is suspended and the *Problem Solver* parameters cannot be changed in the meanwhile. As represented in the same Figure, the *ReplaceBehav-Patt* Global Coordination Operator is used to change the ruling behaviour of all components within the “SteeringInt” PI.

According to what was defined in sections 5.2.2 and 5.2.4, the *ReplaceBehav-*

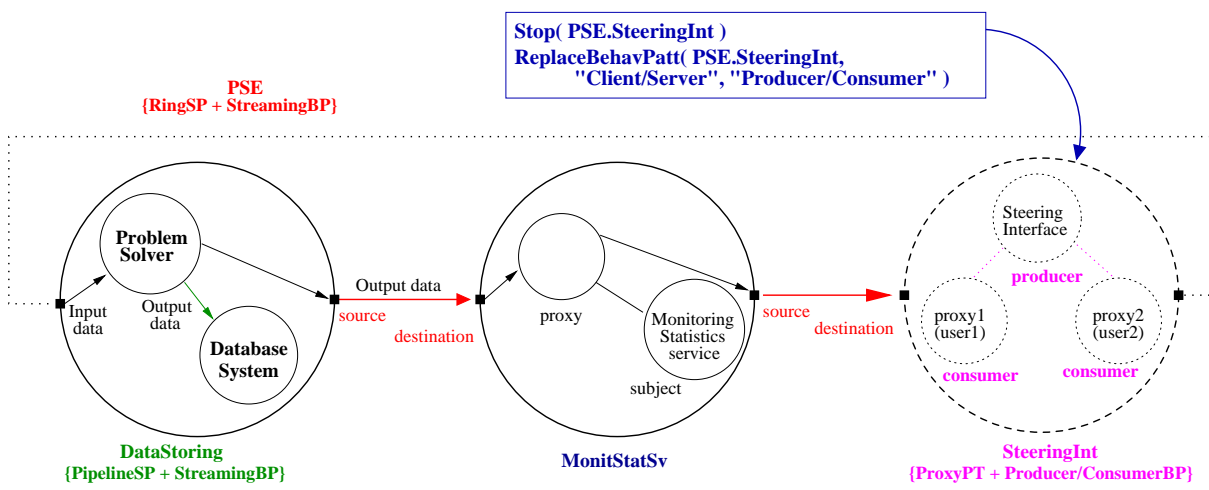


Figure 7.20: Execution suspension of the embedded “SteeringInt” Pattern Instance through the Stop Behavioural Operator and replacement of its Behavioural Pattern.

`Patt(PSE.SteeringInt, “Client/Server”, “Producer/Consumer”)` operator replaces the Behavioural Pattern associated to the “SteeringInt” PI. This operation assumes that the new Behavioural Pattern, i.e. “Producer/Consumer”, associated to the Proxy pattern is also a *Regular PI*, and therefore the new role of each component within the PI is (implementation dependent and) pre-defined. In this case, as presented in Figure 7.20, the “Steering Interface” is defined as the “producer”, and all proxy components will behave as “consumers”.

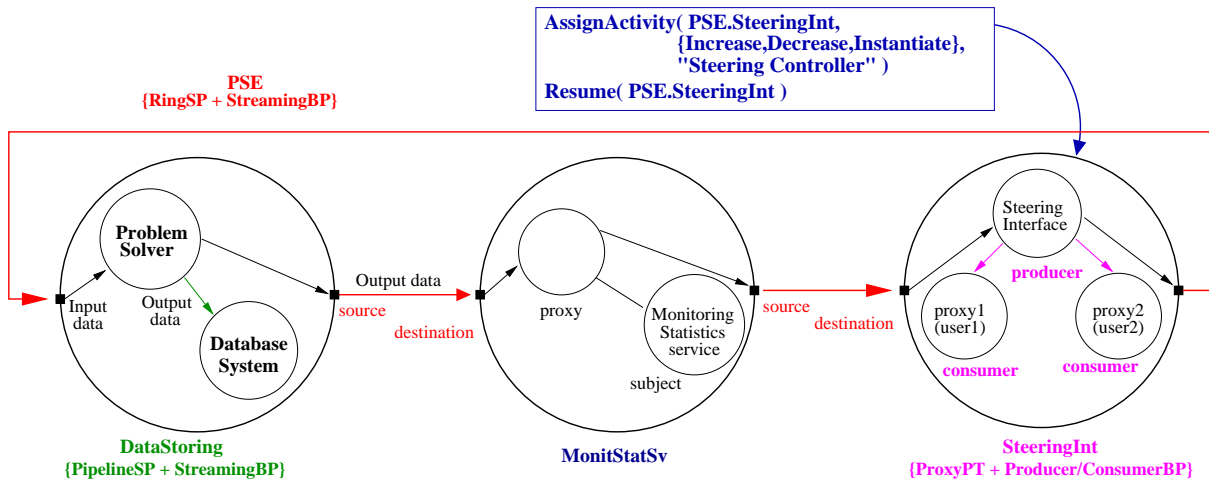


Figure 7.21: Resuming the execution of the embedded “SteeringInt” PI with the definition that the user “Pattern Controller” may manipulate this pattern with the Increase/Decrease and Instantiate operators.

- One of the owners of the “PSE” PI (assigned above through the *DefineOwners* Ownership Operator) uses the *AssignActivity* Ownership Operator to define that the user named “Steering Controller” may operate the “SteeringInt” PI through the *Increase*, *Decrease*, and *Instantiate* operators. This is presented in Figure 7.21. Subsequently, the execution of the *Steering Interface* proceeds through the *Resume(PSE.SteeringInt)*, as displayed at the same Figure.

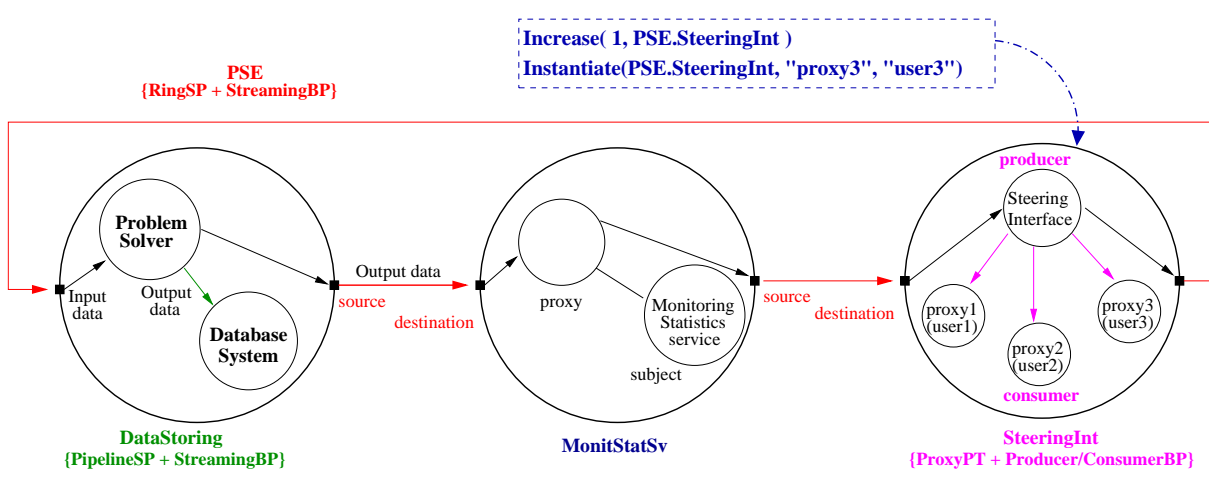


Figure 7.22: Incrementing the number of proxies in the embedded “SteeringInt” providing access to the “Steering Interface” to an extra (passive) user.

3. In case of need to increment the number of remote passive users of the *Steering Interface*, the user “Steering Controller” may then apply the *Increase* Structural Operator with the guarantee that all newly created proxies will behave according to the “consumer” role within the *Producer/Consumer* Behavioural Pattern. This is presented in Figure 7.22 through the application of the *Increase(1, PSE.SteeringInt)*. It is assumed that: a) the new proxy component place-holder (i.e. “proxy3”) is instantiated with a similar component (i.e. “user3” in the Figure) to the other proxies in order to support the access to the “Steering Interface” component; b) the execution of that similar component is automatically launched. This is represented by the *Instantiate(PSE.SteeringInt, “proxy3”, “user3”)* operation.

Please note that although the passive users with proxy access cannot tune the *Problem Solver* through the *Steering Interface*, this one is still directly accessible by the owners of the “PSE” PI that may want to change the application parameters.

Post-mortem Data Analysis

The final reconfiguration illustration of the present PSE example concerns the analysis of the data saved on the *Database system*, once the *Problem Solver* has finished its execution. This development time reconfiguration was previously discussed in section 7.3.1 and illustrated in Figure 7.10.

Starting from a previous configuration for the PSE example, e.g. the one presented in Figure 7.22, the aimed reconfiguration may be achieved in different ways. One option is to not modify the “PSE” Pattern Instance (PI) (e.g. to be saved in a repository):

1. The necessary Pattern Instances for the new configuration are replicated, namely the “Database system”, the “MonitStatSv”, and the “SteeringInt” PIs:

```
1: Replicate(1,PSE.DataStoring.DatabaseSystem, ``DatabaseSystemPM`` )
2: Replicate( 1, PSE.MonitStatSv, ``MonitStatSvPM`` )
```

```
3: Replicate( 1, PSE.SteeringInt, ``SteeringIntPM`` )
```

2. A three stage pipeline ruled by the Streaming Behavioural Pattern is generated in order to connect the replicated PIs:

```
4: Create( PipelineSP, ``PSEPM``, 3)
5: DefineBehavPatt( PSEPM, ``Streaming`` )
```

It is assumed that data flows from the first stage of the pipeline (named “cph1”) to the second (i.e. “cph2”), and from this one to the third stage (i.e. “cph3”).

3. In order to process the *Problem Solver*’s generated data saved at the *Database system*, its ruling behaviour is changed to the *Client/Server* Behavioural Pattern. In this way, the principal Database located at the nucleus of the star-based “DatabaseSystemPM” becomes responsible for retrieving the data spread at the distributed secondary databases defined as the satellites of the Star pattern ²:

```
6: ReplaceBehavPatt(DatabaseSystemPM, ``Master/Slave``, ``Client/Server``)
```

As presented in Figure 7.23, the principal Database (at the nucleus) is annotated with the “client” role, and all secondary Databases at the satellites are defined as “servers”.

4. The replica “DatabaseSystemPM” is embedded in the first stage of the “PSEPM”, the replica “MonitStatSvPM” in the second, and the “SteeringIntPM” in the third:

```
7: Embed( DatabaseSystemPM, PSEPM, ``cph1`` )
8: Embed( MonitStatSvPM, ``cph2`` )
9: Embed( SteeringIntPM, ``cph3`` )
```

5. The *Start(PSEPM)* Behavioural Operator is used to launch the execution of all components/services in the application. It is assumed that this operator acts recursively upon the “PSEPM” Hierarchical PI launching the execution of all components/services.

The actions enumerated above lead to the configuration presented in Figure 7.23.

Another option to perform a configuration for the post-mortem analysis of the generated data is similar to first option, except that the necessary PIs are extracted from the “PSE” PI and this one is itself deleted:

```
1: Extract( DatabaseSystem, PSE.DataStoring, ``cph2`` )
2: Extract( MonitStatSv, PSE, ``cph2`` )
3: Extract( SteeringInt, PSE, ``cph3`` )
```

²Under the assumption that the *Master/Slave* Behavioural Pattern would also support the retrieval of data from the secondary databases at the satellites, the behavioural change would not be necessary.

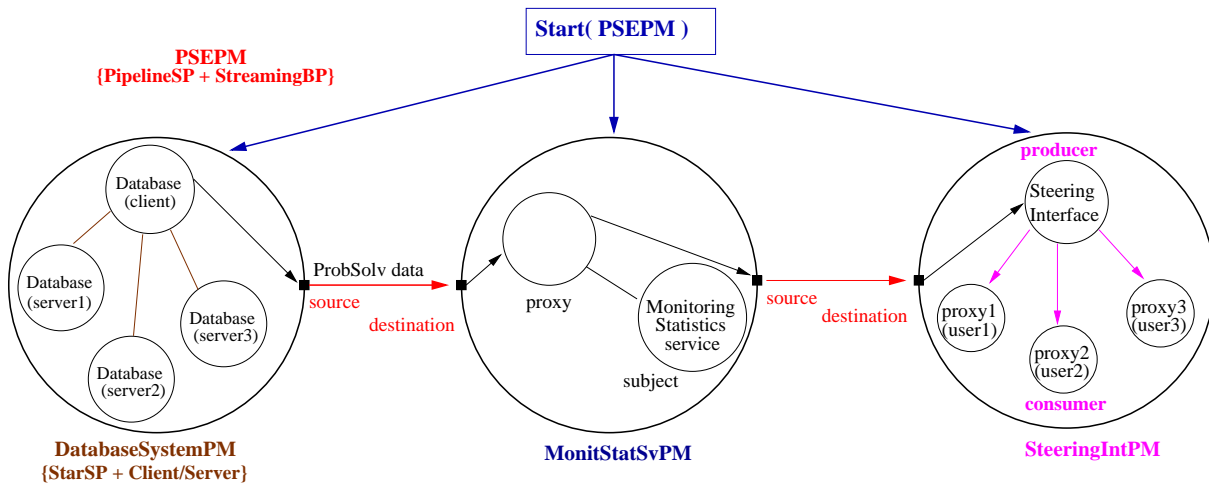


Figure 7.23: New configuration for analysis of the PSE generated data formerly saved at the Database system.

```

4: Eliminate( PSE )
5: Create( PipelineSP, ``PSEPM'', 3)
6: DefineBehavPatt( PSEPM, ``Streaming'' )
7: ReplaceBehavPatt( DatabaseSystem, ``Master/Slave'', ``Client/Server'' )
8: Embed( DatabaseSystem, PSEPM, ``cph1'' )
9: Embed( MonitStatSv, ``cph2'' )
10: Embed( SteeringInt, ``cph3'' )
11: Start( PSEPM )

```

Finally, an hypothetical option based on the *Reshape* Structural Operator could be considered to simplify the present reconfiguration of the “PSE” PI. An operator sequence using that operator could be:

```

1: Extract( DatabaseSystem, PSE.DataStoring, ``cph2'' )
2: ReplaceBehavPatt( DatabaseSystem, ``Master/Slave'', ``Client/Server'' )
3: Eliminate( PSE.DataStoring )
4: Reshape( PSE, PipelineSP )
5: Embed( DatabaseSystem, PSE, ``cph1'' )
6: Start( PSE )

```

However, such reconfiguration is not possible due to the restriction that the *Reshape* operator cannot be applied to CISP, SB-PTs, and PIs, as discussed throughout section 5.2. In this particular example, it would be necessary to establish, upon calling the *Reshape(PSE, PipelineSP)* operator, that although the same Behavioural Pattern, i.e. *Streaming*, would be used for the new (Pipeline) structure: a) the “cph1” element in the original Ring Structural Pattern would become the first stage of the new pipeline-based structure for the “PSE” PI, the “cph2” would become the second pipeline stage, etc. ; and b) the data flow would continue to proceed from the first pipeline stage to the second, and from this one to the third. In a simple analysis, this would correspond to break the ring connection between the stage containing the “SteeringInt”

and the first stage (formerly containing the “DataStoring” PI). Although the usage of the *Re-shape* operator upon CISP, SB-PTs, and PIs raises a high number of different possible transformation mappings, we intend to make that study in future versions of our work.

Addition of an extra Component

Finally, as an additional example of a development time reconfiguration, we may assume that it is necessary to add a *Visualisation Component* to the configuration. Specifically, this component is to display all the information generated by the *Problem Solver*, i.e. the *Visualisation Component* has to be directly connected to the *Problem Solver*, and not to the *Monitoring Service*.

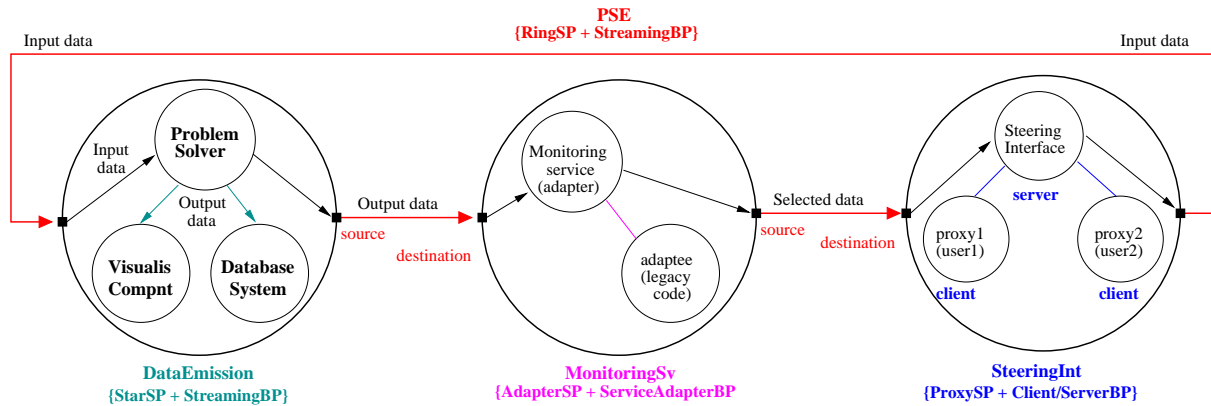


Figure 7.24: A new component is directly connected to the Problem Solver. This configuration is based on the one presented in Figure 7.17.

One way to achieve such reconfiguration is to define a Star Structural Pattern, also combined with the *Streaming* Behavioural Pattern, where the *Problem Solver* at the “nucleus” sends data to both “satellites”, namely the *Database system* and the *Visualisation Component*. This is represented in Figure 7.24.

Such reconfiguration may be achieved with the following operator sequence:

```
1: Create( StarSP, ``DataEmission``, 3 )
2: Extract( DatabaseSystem, PSE.DataStoring, ``cph2`` )
3: Embed( DatabaseSystem, DataEmission, ``satellite2`` )
4: Instantiate( DataEmission, ``satellit1``, ``VisualisCompnt`` )
5: Instantiate( DataEmission, ``nucleus``, ``ProblemSolver`` )
6: Replace( PSE.DataStoring, DataEmission )
```

We assume that the instantiation above of the “nucleus” component place holder (CPH) with the *Instantiate(DataEmission, “nucleus”, “Problem Solver”)* operator refers exactly the same *Problem Solver* used so far in the several PSE examples. To instantiate the remaining CPHs of the newly created pattern, i.e. “DataEmission”, the operations are similar to the ones described so far. Subsequently, the “DataEmission” PI takes the place of the “DataStoring” PI as the first stage of the ring-based PSE through the *Replace(PSE.DataStoring, DataEmission)* operator.

7.4 Skeleton Modelling

As previously described in section 2.3.1, skeleton-based approaches have been providing for already some time a way to compose parallel programs by reusing basic recurrent parallel structures without the need to code their typical interactions. In their classic definition, skeletons are, in general, polymorphic, higher-order functions with pre-packaged implementations specifically tuned for (high-performance) parallel systems. Several functional programming languages defined skeletons as their basic constructor [152,155,167] but without allowing *skeleton nesting*, whereas other works support an adequate composition of skeletons to form more complex applications [151,157].

With time, skeleton usage has been naturally ported from the parallel programming domain into Grid-based environments as a way to simplify programming [259–261]. Although being proven as a powerful concept to model parallel interactions, the classical skeleton definition lacks the expressiveness to express design issues in Grid environments, which are better defined, in our opinion, through patterns as the ones we propose in this dissertation. Nevertheless, it is possible to find similarities between the interaction structures underlying typical skeletons, and some combinations of Structural Patterns (e.g. topological such as Pipeline or Star patterns) with Behavioural Patterns (e.g. Streaming or Master/Slave patterns). As such, our goal in this section is simply to provide an example of how our patterns and operators may be used to model the interactions inherent to some typical skeletons. This is done in the context of a particular skeleton-based language, namely the *Pisa Parallel Programming Language (P3L)* [154]. The *P3L* language already provides *skeleton nesting* which can also be represented in our model, namely through the *Embed* operation.

A general introduction to *P3L* was previously presented in section 2.4.1. The next sub-section describes how to configure some of its basic skeletons through our model entities, whereas the following sub-section presents the modeling of one *P3L* example.

7.4.1 Mapping P3L Skeletons to Structural and Behavioural Patterns

The *P3L* composition language is based on *Data Parallel skeletons (DPs)*, *Task Parallel skeletons (TPs)*, and *Control Parallel skeletons (CPs)*, making it suitable for mixed task and data parallelism applications. DPs abstract array partition and alignment of dense multi-dimensional array structures, and include the *map*, *reduce*, and *comp* skeletons. TPs exploit parallelism at coarse level, namely between the execution of instances of DPs and for a stream of homogeneous independent input data, and include the *pipeline* and *farm* skeletons. DPs and TPs are the configuration constructors, whereas CPs do not change the parallel structure of the application. Namely, the *seq CP* abstracts sequential code to instantiate truly parallel skeletons, and the *loop CP* controls skeleton execution iteration.

Control and Task Parallel Skeletons

Figure 7.25 presents three skeletons defined by the *P3L* language that represent different forms of execution control. The first skeleton is the *sequential control skeleton (seq CP)* (a) which represents a module that encapsulates code written in a sequential language and that may be used

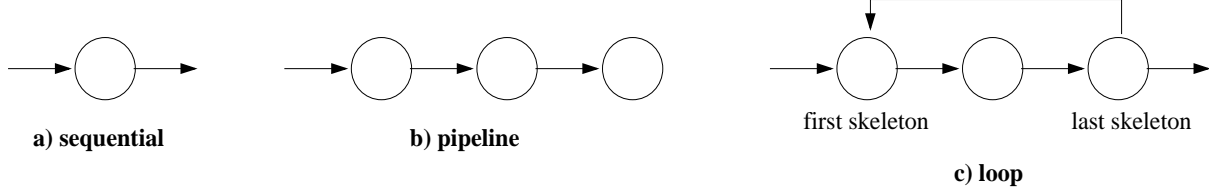


Figure 7.25: Two Control Parallel skeletons: a) sequential and c) loop; and one Task Parallel skeleton: b) pipeline.

from the P3L language. The in/out interfaces are well defined in the module and it is used to instantiate truly parallel skeletons. In our model, a *seq CP* corresponds to the instantiation of a component place-holder (CPH) in a Structural Pattern Template (S-PT) to a program executing sequentially. For example, in the model implementation to the Triana PSE described in Chapter 6, a CPH is instantiated to a Triana “tool/unit” whose interfaces are also well defined through input and output ports.

The second skeleton in the Figure (b) is the *pipeline Task Parallel skeleton* which represents a sequence of skeleton instances (i.e. *tasks* in P3L) executing concurrently. The *pipeline TP* skeleton defines a sequence of independent stages of computation, and it may represent a sequence of both DP and/or TP skeletons. In our model, the *pipeline TP* may be directly mapped to a *Pattern Template (SB-PT)* resulting from a *Pipeline S-PT* where data and control flow between stages may be ruled by the *Streaming Behavioural Pattern*. Moreover, and due to our *Embed Structural Operator*, each stage in that SB-PT can enclose itself a SB-PT to model a DP or a TP in P3L.

The third skeleton in Figure 7.25 (c) is the *loop Control skeleton* which iterates skeleton composition until some condition is verified (a single input may cause several executions of the skeletons controlled by the loop). The representation of the loop skeleton in our model may be done in two ways. On one hand, the application of the *Repeat Behavioural Operator* to a Pipeline-based SB-PT (e.g. ruled by the *Streaming BP*) could represent the *loop* skeleton, if the loop condition is to be a number defining how many times the skeleton sequence has to run, and that number does not depend on data produced by the last skeleton instance in the sequence.

On the other hand, the cyclic execution control of a sequence of skeletons defined by the loop control skeleton may be structurally represented by the *Ring S-PT*. First, the application of the *Streaming Behavioural Pattern* to the inner stages that represent the skeleton sequence may represent the data and control flows between those skeletons. Second, the structural connection from the last CPH back to the first CPH (i.e. the one that closes the cycle between the last skeleton in the sequence and the first one in the sequence) is to represent (at least) a control flow allowing triggering another execution of the skeleton sequence.

In the Triana-based implementation of our model, this is possible through the structural connection to special ports named *trigger nodes* (as described in Chapter 6) which allows activating the execution of the unit they belong to. Although that ring-based SB-PT could model a circular execution, the loop skeleton semantics requires the evaluation of a condition. Such evaluation could be performed by an extra stage placed between the one that represents the last

skeleton in the sequence and the stage that represents the first skeleton. In this way, it is possible to define the loop condition based on data produced by the last skeleton in the sequence, that upon positive evaluation would result on the triggering of another execution.

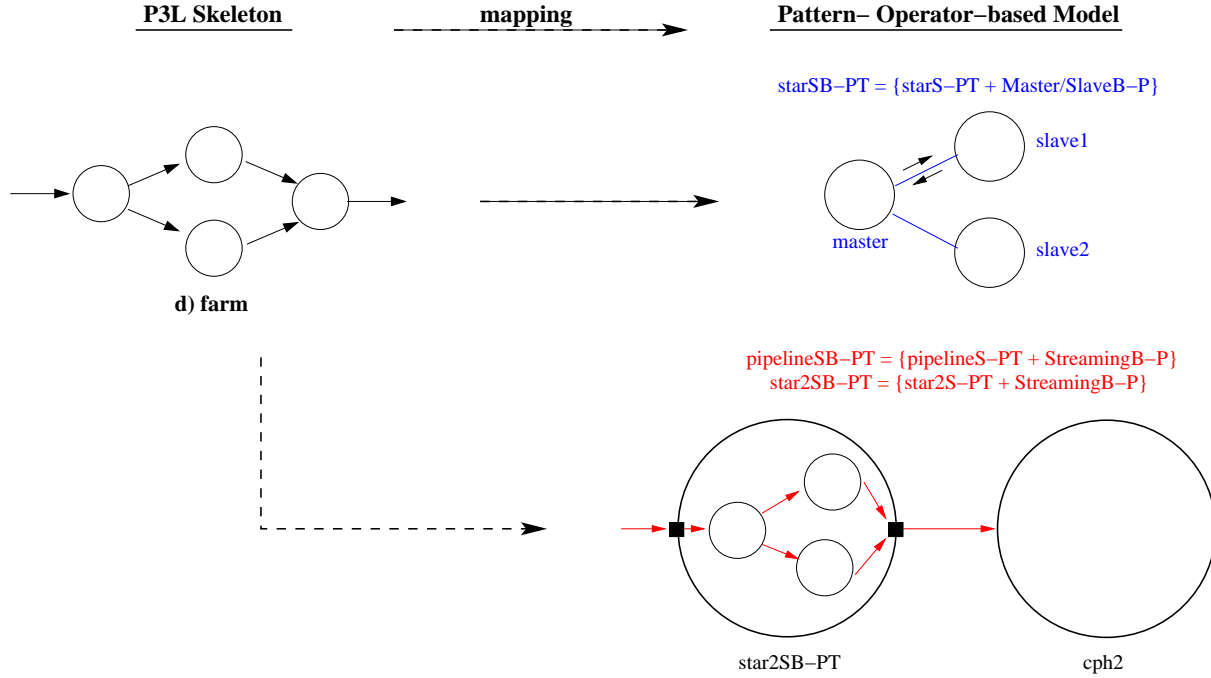


Figure 7.26: Modelling the *farm* Task Parallel skeleton.

The *farm* skeleton is another Task Parallel skeleton which is presented in Figure 7.26 along with two possible ways for its modeling through Structural and Behavioural Patterns and Operators. The *farm* TP skeleton replicates a skeleton (a DP or TP) in a pool of identical copies – the *workers* – that run in parallel. Each *worker* computes independent data items, i.e. for each data item, the controller of the *farm* TP skeleton selects which *worker* will compute that data.

In the example of a *farm* TP in the Figure (d), the first element in the skeleton, i.e. the *controller* (depicted in the leftmost position), sends data to be processed by one of the next two elements (the *workers*) and according to an implementation dependent scheduling rule. The results are gathered by the last element (the rightmost one). In our model, this *farm* TP skeleton may be configured in two ways, although the first option described next is more adequate in our opinion. Namely, the first option considers that the controller itself gathers all the computed results. Consequently, the skeleton may be modelled through a *Star* S-PT combined with the *Master/Slave* Behavioural Pattern, as presented on the upper part of the right-hand side of Figure 7.26. The “nucleus” of the *starSB-PT* has the “master” role representing the *controller* in the *farm* TP skeleton, and the “satellites” are the “slaves” i.e. the *workers*.

The second configuration option of the *farm* closely follows its configuration, i.e. the *controller* does not collect the computed results, but they are collected by another entity as presented in the Figure 7.26. To build such configuration, two SB-PTs are necessary. The first SB-PT named “*pipelineSB-PT*” results from a two stage *Pipeline* S-PT combined with the *Streaming* Behavioural Pattern. The second SB-PT named “*star2SB-PT*” combines a *Star* S-PT (containing two satellites) also with the *Streaming* Behavioural Pattern. This second SB-PT represents the

interactions between the controller of the *farm* (i.e. the “nucleus”) and the workers (i.e. the “satellites”), and it is up to the controller to select which worker will compute the input data. The “star2SB-PT” represents the first stage to be included into the first component place-holder of the “pipelineSB-PT”, and the second stage of this pipeline represents the entity that gathers the computed results. Therefore, the *Embed* Structural Operator is used to make that inclusion. The following operator sequence defines one possible way of building this second modelling option of the *farm* skeleton:

```
Create( PipelineSP, ``farmPT``, 2 )
Create( StarSP, ``starSB-PT``, 3 )
DefineBehavPatt( farmPT, ``Streaming`` )
Embed( starSB-PT, farmPT, ``cph1`` )
DefineBehavPatt( farmPT.starSB-PT, ``Streaming`` )
```

Please note that, in this operator sequence example, it is assumed that the embedding operation of a star into an element of a pipeline implies establishing structural connections from all satellites in the “starSB-PT” into the second stage of the “farm” pipeline. Those connections represent the data flow of the computed results from all workers into the entity in the second stage that collects that computed data. Moreover, it also assumed that the application of the *DefineBehavPatt*(*farmPT.starSB-PT*, “Streaming”) results on a data flow from the “nucleus” of the “starSB-PT” to the satellites.

Data Parallel Skeletons

Data Parallel skeletons (DPs) in *P3L* provide abstractions to hide a few common configurations for parallel processing of multi-dimensional array structures. The actual mapping into a disjoint set of processors is implementation dependent. *P3L* DPs include the *map*, *comp*, and *reduce* skeletons [154].

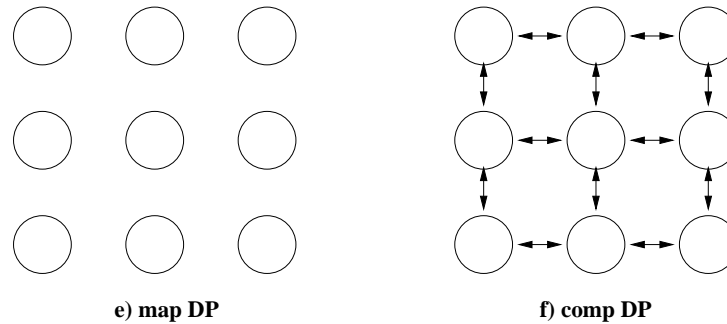


Figure 7.27: The “map” and “comp” Data Parallel skeletons.

Figure 7.27 represents the *map* (e) and the *comp* (f) Data Parallel skeletons (DPs) in *P3L*. In general, the *map skeleton* takes a function and a data structure as arguments and applies the function to each element of the data structure. With the *map* DP in *P3L*, the same computation is replicated and run in parallel at several nodes to process all elements of a (possibly nested) data structure. The *comp* skeleton, in turn, combines several data parallel stages modelling common functional composition. Although these two skeletons are not directly supported by

the patterns in our model, they can be included in a pattern-based configuration through the *Adapter* Structural pattern. Available code, or for instance a service, implementing those two DPs can be interfaced by the *Adapter* pattern combined with an adequate Behavioural Pattern.

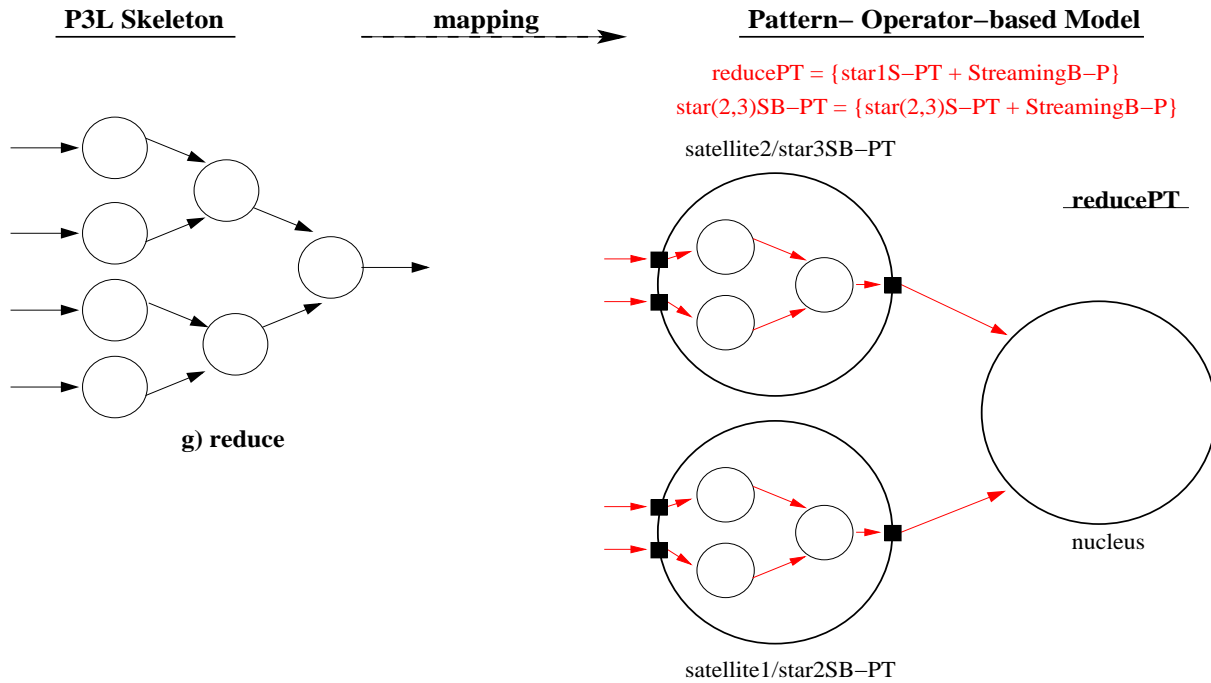


Figure 7.28: Modelling the reduce Data Parallel skeleton.

The left-hand side of Figure 7.28 represents a third Data Parallel skeleton in *P3L*, namely the *reduce* skeleton. The *reduce* skeleton allows the application of an associative binary operator to the elements of an array and this reduction operation is executed in parallel resulting on the “sum” of all elements. An available implementation of the *reduce* skeleton may as well be included in a pattern-based configuration through the *Adapter* pattern.

Nevertheless, the right-hand side of Figure 7.28 presents a possible way of building a similar configuration based on the *Star* Structural Pattern. The basic pattern is a *Star* S-PT with two satellites representing one reduction operation which is to be operated at the “nucleus”. The *reduce* topology can then be represented by the hierarchical combination of similar replicas of that basic S-PT. The “*reducePT*” in the picture represents the outmost *Star* PT which contains two other *Star* PTs, each one embedded in one of its “satellites”. As displayed in the Figure, the *Streaming* Behavioural Pattern is again used to represent the data flow. The following operator sequence defines a possible way to build the “*reducePT*” Pattern Template:

```
Create( StarSP, ``reducePT'', 3 )
DefineBehavPatt( reducePT, ``Streaming'' )
Replicate( 2, reducePT, { ``star2SB-PT'', ``star3SB-PT'' } )
Embed( star2SB-PT, reducePT, ``satellite1'' )
Embed( star3SB-PT, reducePT, ``satellite2'' )
```

In this operator sequence we make two assumptions. First, in this application of the *Streaming* Behavioural Pattern to a *Star* S-PT, we assume that data flows from the satellites to the

“nucleus” of the Star, contrary to other examples where data flows from the “nucleus” to the satellites³.

Second, the embedding of both inner Stars SB-PTs, i.e. “star2SB-PT” and “star3SB-PT”, implies establishing structural connections with their enclosing satellites, i.e. “satellite1” and “satellite2”, respectively, so that incoming data to these satellites and thereafter processed at “star2SB-PT” and “star3SB-PT” is to flow to the “nucleus” of the “reducePT” pattern.

Skeleton Nesting and Execution in P3L

The *P3L* provides *skeleton nesting* where:

- *Data Parallel skeletons (DPs)* may be nested in other DPs;
- *Task Parallel skeletons (TPs)* may be nested in other TPs;
- DPs may be also nested in TPs, but TPs cannot be nested in DPs;
- *Control Parallel skeletons (CPs)* may be nested either on TPs, DPs, and other CPs, since they do not change the parallel structure of the application.

After the definition of the overall configuration of an application based on *P3L* skeletons, their instantiation originates *tasks* (either *sequential* or *parallel* tasks) to be executed. DP tasks are executed in parallel, and when an input item is computed in a DP task, its data parallel skeletons are computed in all the processors assigned to that DP task, and intermediate results are kept distributed. Moreover, the executions and interactions of independent DP tasks are controlled by task parallel skeletons. Namely, results produced by a DP task are forwarded to the next (usually) DP task, according to the global structure defined by TPs skeletons (e.g. two DP tasks in two stages of a pipeline TP). Data flow implementation is hidden from the programmer, e.g. the necessary input fetching and sending of results can be delegated to a centralised controller or to an appropriate subset of the processes in a DP task. The termination of a *P3L* program happens when the end of the input stream is detected by all processes in the program.

The next sub-section presents a possible configuration for an example in *P3L* that includes skeleton nesting.

7.4.2 Modelling a P3L Example

In [151] a typical example of *P3L* is described which is presented in Figure 7.29. This particular *P3L* computation is composed of four DP tasks (DP1, DP2, DP3, and DP4 implement the “map” Data Parallel skeleton) interacting according to the composition of a farm and a pipeline Task Parallel skeletons. All tasks in DP1 (first stage of the pipeline) run in parallel and the produced results are sent to the next second stage in the pipeline where, in turn, all tasks in DP2 process the income data and also send the results to the next stage. Stage three in the pipeline is controlled by a farm TP task, where the controller element in the farm TP sends data to one of the

³As described in section 5.2.2, this is a case where the same B-P can be applied to a S-PT in two different ways, which need further clarification to distinguish both situations.

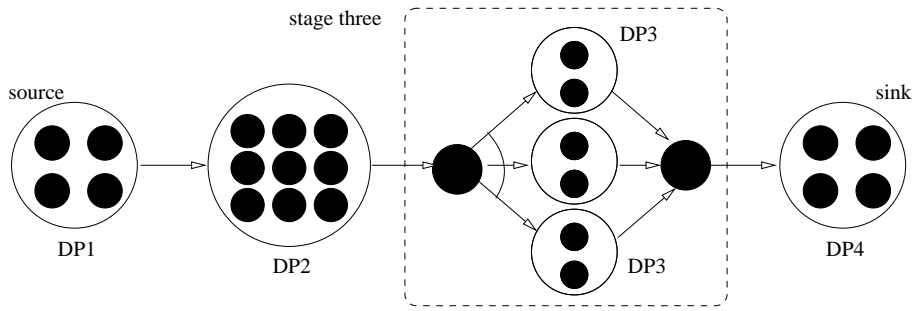


Figure 7.29: A P3L example [151] composed of four DP tasks interacting according to the composition of a farm TP and a pipeline TP.

DP3 replicas. Results are merged at the farm task element acting as a receiver, and are then sent to the final stage of the pipeline (DP4 task).

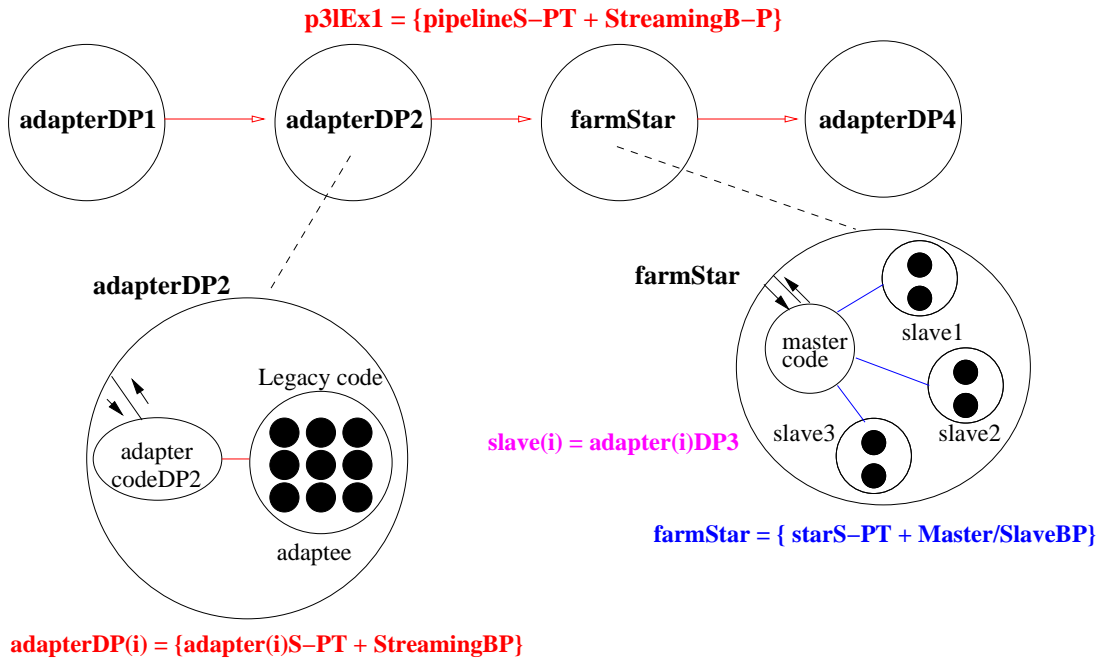


Figure 7.30: One case of modelling the P3L example in Figure 7.29 using the Pattern/Operator model.

One way of modelling of the example of Figure 7.29 through the Pattern/Operator model is shown in Figure 7.30. First, a four-stage Pipeline S-PT combined with a Stream Behavioural Pattern, may be used to represent the pipeline TP skeleton (“p3lEx1” in the Figure).

Second, the modelling of the farm TP skeleton may be supported by a Star S-PT with three satellites combined with the *Master/Slave* Behavioural Pattern (“farmStar” in Figure 7.30) – each satellite is a “slave” within this B-PT and the nucleus has the “master” role. As explained in the previous sub-section (see Figure 7.26), such modelling considers that the “master” selects to which “satellite” each independent data input should be sent to, and also collects the results.

Third, all DPs in the example in Figure 7.29 may be considered as being supported by legacy code specific of parallel programming that implements the “map” skeleton. Namely, that legacy code represents the replication and parallel execution of the same computation

over several nodes to process all elements of a (possibly nested) data structure.

In our model, such legacy codes may be made accessible in the configuration through Adapter S-PTs combined with the *Streaming* Behavioural Pattern – the “adapter” element interfaces the legacy code, i.e. the “adaptee” element in the pattern, and input data flows from the “adapter” to the legacy code whose results are again collected by the “adapter”. The notation in Figure 7.30 defines that the “adapterDP1” represents the DP1 skeleton in Figure 7.29, “adapterDP2” represents the DP2 skeleton, etc. The DP3 skeleton, in particular, is replicated three times by the farm skeleton. These replicas are named “adapter1DP3”, “adapter2DP3”, and “adapter3DP3” in Figure 7.30, and they instantiate, respectively, the “slave1”, “slave2”, and “slave3” elements in “farmStar”.

The operator sequence that may be used to build and execute the above configuration will be described in three parts.

Operator sequence, part one Represents the creation and composition of the SB-PTs that define the Task Parallel skeletons in the example:

```
1: Create( PipelineSP, ``p3lEx1'', 4 )
2: DefineBehavPatt( p3lEx1, ``Streaming'' )
3: Create( StarSP, ``farmStar'', 4 )
4: DefineBehavPatt( farmStar, ``Master/Slave'' )
5: Embed( farmStar, p3lEx1, ``cph3'' )
6: Instantiate( p3lEx1.farmStar, ``nucleus'', ``mastercode'' )
```

- *Steps 1 and 2* create the Pattern Template that models the pipeline TP skeleton in the example. It is assumed that the application of the *Streaming* Behavioural Pattern defines that data flows from left to right, i.e. from the first stage in the pipeline (left-most one), which is named “cph1” thereafter, to the second stage (“cph2”), etc.
- As for *steps 3 and 4*, they create the Pattern Template that models the farm TP skeleton – a star with three satellites is created, and upon the application of the *Master/Slave* Behavioural Pattern, the satellites are annotated with the “slave” role, and the nucleus becomes the “master”. The master sends each data frame to one of the slaves (where each slave is to be a replica of each other), and collects the results.
- *Step 5* defines the embedding of the “farmStar” SB-PT into the third stage (“cph3”) of the “p3lEx1” modelling skeleton nesting. *Step 6* defines the code that implements the master, namely through the *Instantiate* operator.

Operator sequence, part two Represents the creation of the necessary Adapter-based SB-PT patterns to represent two Data Parallel skeletons in the example, namely DP2 and DP3. DP3 is also replicated to instantiate the defined satellites/slaves in the farm skeleton configuration:

```
7: Create( AdapterSP, ``adapterDP2'' )
8: Instantiate( adapterDP2, ``adapter'', ``adaptercodeDP2'' )
9: Instantiate( adapterDP2, ``adaptee'', ``legacycodeDP2'' )
```

```

10: DefineBehavPatt( adapterDP2, ``Streaming`` )
11: Embed( adapterDP2, p3lEx1, ``cph2`` )
12: Create( AdapterSP, ``adapter1DP3`` )
13: DefineBehavPatt( adapterDP3, ``Streaming`` )
14: Instantiate( adapter1DP3, ``adapter``, ``adaptercodeDP3`` )
15: Instantiate( adapter1DP3, ``adaptee``, ``legacycodeDP3`` )
16: Replicate( 2, ``adapter1DP3``, { ``adapter2DP3``, ``adapter2DP3`` } )
17: Embed( adapter1DP3, p3lEx1.farmStar, ``satellite1`` )
18: Embed( adapter2DP3, p3lEx1.farmStar, ``satellite2`` )
19: Embed( adapter3DP3, p3lEx1.farmStar, ``satellite3`` )

```

- Steps 7 to 11 define the PI named “adapterDP2” that models DP2 and which is embedded in the second stage of the pipeline. As defined before, the behaviour coordinating the elements in the adapter is to be the *Streaming* BP (step 10). It is assumed that data flows from the adapter element (“adaptercodeDP2” in Figure 7.30) to the adaptee element (“legacycodeDP2” in the Figure), and also back from the adaptee to the adapter.

Moreover, it is also assumed that the adapter is structurally connected to both the previous stage and the next stage in the pipeline – data flows from the previous stage, namely “adapterDP1”, to the “adaptercodeDP2” element, and results obtained by the “adaptercodeDP2” from the “legacycodeDP2” flow to the next stage, namely to “farmStar”.

Please note that the adaptation of all DP(i) in this example is coordinated by the *Streaming* BP, and similar assumptions to the ones made for “apaterDP2” are again presumed for all DP(i).

- Steps 11 to 14 create three similar patterns to represent the replicated DP3 in the example (Figure 7.29).
- In steps 15 to 17, the replicas (“adapter1DP3” .. “adapter3DP3”) are embedded in the satellites of the star-based pattern that models the farm skeleton.

Operator sequence, part three Finally, the last sequence represents the modelling of the remaining DPs, namely DP1 and DP4, as well as the execution activation of the final configuration. It is assumed that the necessary implementation code (named “adaptercode” bellow) to interface both legacy codes representing DP1 and DP4 is similar:

```

20: Create( AdapterSP, ``adapterDP1`` )
21: DefineBehavPatt( adapterDP1, ``Streaming`` )
22: Instantiate( adapterDP1, ``adapter``, ``adaptercode`` )
23: Replicate( 1, adapterDP1, ``adapterDP4`` )
24: Instantiate( adapterDP1, ``adaptee``, ``legacycodeDP1`` )
25: Instantiate( adapterDP4, ``adaptee``, ``legacycodeDP4`` )
26: Embed( adapterDP1, p3lEx1, ``cph1`` )
27: Embed( adapterDP4, p3lEx1, ``cph4`` )
28: Start( p3lEx1 )

```

The common configuration (named “adapterDP1”) to both DP1 and DP4 is built in steps 20 to 22. Step 23 replicates that configuration under the name “adapterDP4”. The instantiation to the adequate legacy codes is done in steps 24 and 25, and the embedding of the final PIs is done in steps 26 and 27.

Finally, the execution is launched through the *Start(p3lEx1)* operator. In this case, it is assumed that this Behavioural Operator acts recursively upon all embedded patterns to trigger the execution of all involved elements.

7.4.3 Reconfiguring the P3L Example

This sub-section presents two possible reconfigurations of the P3L example described in Figure 7.4.2. The first case is based on the pattern-based configuration described in the previous section. The second example is based on a second P3L example described in [151] which is itself related to the first P3L example in Figure 7.4.2.

First Reconfiguration Case

The first example highlights the usefulness of our model considering the possible execution of the described P3L computation in a Grid environment. The following description highlights how the first pattern-based modelling depicted in Figure 7.30 may be reconfigured to that purpose. Please note that although the described P3L computation is typical of high-performance systems where performance is a main issue, we consider the situation where its porting to a Grid environment is beneficial in terms of reuse of available Grid services. Such situation might be useful if, for example, high-performance computational resources are not available locally. Therefore, this pattern-based modelling of the above P3L computation in a Grid environment makes a few assumptions. Specifically:

1. We consider that the entities forming the stages of the pipeline in the example (Figure 7.30) are distributed on the Grid. Although we define that the attached Behavioural Pattern in the following reconfiguration is still the *Streaming BP*, the *Producer/Consumer BP* could also represent the flow of data between the distributed pipeline stages, where data between stages is saved so that it is consumed as soon as possible.
2. We also consider that the data parallel computations DP2 and DP4 may be accessible as services on the Grid. As such, the “adapterDP2” and “adapterDP4” patterns, at the second and fourth stages in Figure 7.30, are now replaced, respectively, with the “GridServiceDP2” and “GridServiceDP4” patterns, as presented in Figure 7.31. Each of these two new patterns consists of the combination of an Adapter S-PT with the *Service Adapter Pattern* Behavioural Pattern (see section 3.2.4). “GridServiceDP2” and “GridServiceDP4” still interface the legacy codes supporting the replicated parallel execution of a computation but that are now accessible as (Grid) services. The reconfiguration may be done by:
 - (a) checking the full compatibility of the “GridServiceDP2” and “GridServiceDP4” with the original patterns through the *IsCompatible(P1, P2)* Inquiry Operator (see section

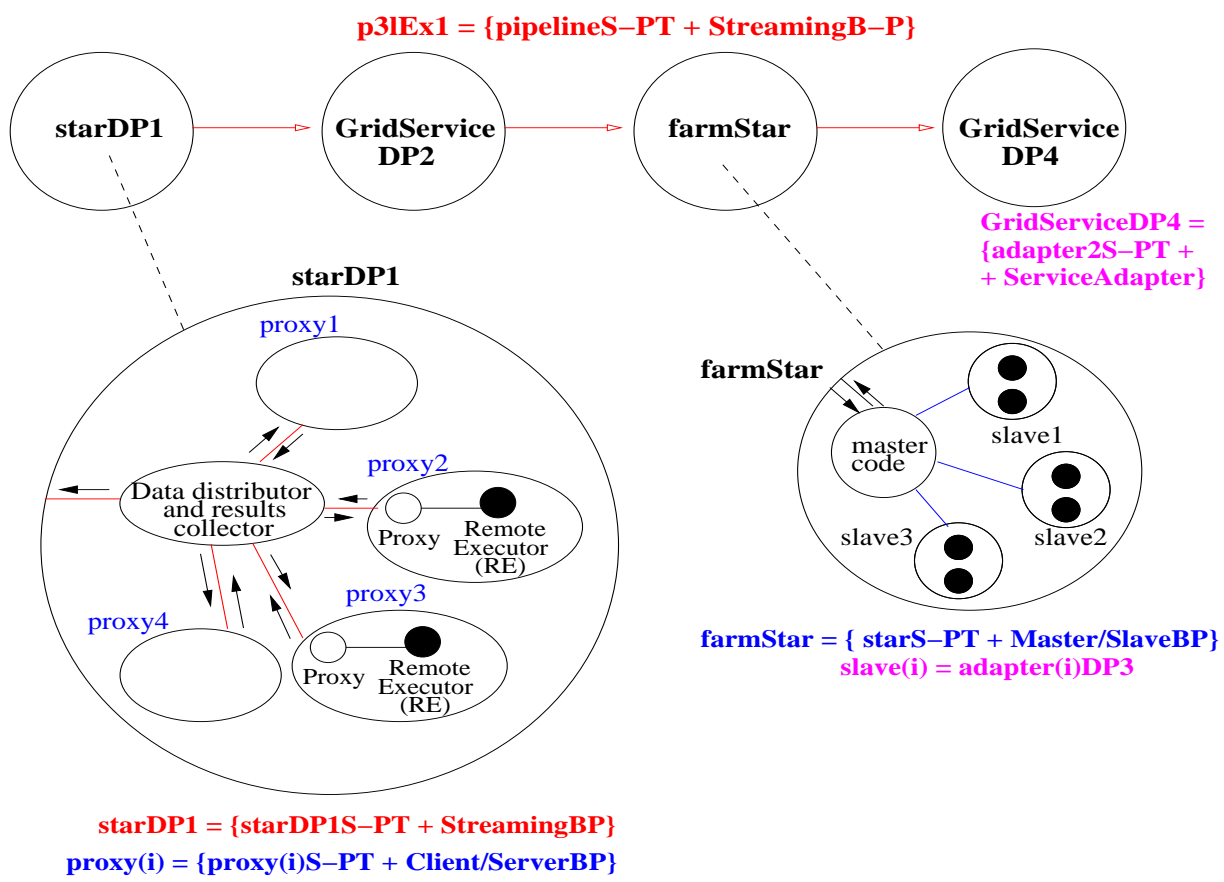


Figure 7.31: One possible reconfiguration of the pattern-based example in Figure 7.30

3.3.3) to guarantee that they are functionally identical: *IsCompatible*(GridServiceDP2, p3LEx1.adapterDP2), and *IsCompatible*(GridServiceDP4, p3LEx1.adapterDP4);

- (b) replacing the original patterns to the “GridServiceDP2” and “GridServiceDP4” patterns through the *Replace* Structural Operator which is applied to the patterns as first class entities (as explained in sections 5.2.2 and 5.2.4): *Replace*(p3LEx1.adapterDP2, GridServiceDP2) and *Replace*(p3LEx1.adapterDP4, GridServiceDP4).
3. In the “farmStar” pattern (third pipeline stage in Figure 7.30, and also in Figure 7.31) the satellites/slaves may also be distributed in the Grid, i.e. each of these slaves “adapter1DP3”.. “adapter3DP” representing a replica of the DP3 may be supported by computational resources at different locations. The master sends each data frame received from the “adapterDP2” to one of the slaves, and all collected results are subsequently sent by the master (“mastercode” in the Figure) to the “adapterDP4”. We again assume that each “adapter(i)DP3” provides access to replicated parallel computations supporting the “map” Data Parallel skeleton.
4. Finally, we define a different configuration for the first stage of the pipeline (which was formerly represented by the “adapterDP1” in Figure 7.30) based on the assumption that the data to be processed by the parallel computations in DP1 is very huge. As such, the data to be processed might be already spread throughout four distributed “repositories” within the Grid, where each repository has associated high performance execution support. Therefore,

each of the four replicated computations composing DP1 in the original example would be executed locally at the each repository in order to process a sub-set of the data.

Taking in consideration this fourth assumption for the example, a Star SP might be used to represent those kind of repositories, where the function of the entity at the “nucleus” element in the star would be to collect the produced results, that would be subsequently forwarded to the second stage of the pipeline (i.e. to “adapterDP2”). In case some additional data besides the existing in the repositories would be necessary, the function of the nucleus would also be to send that extra data to one ore more repositories. Moreover, we can also assume that the communication to each distributed repository might be done through local proxies, and could therefore be represented by instances of the Proxy Structural Pattern.

This new configuration for the first stage is also presented in Figure 7.31. The Star SP is named “starDP1” and the entity at the nucleus is named “Data distributor and results collector”. The satellites are designated “proxy1”.. “proxy4” and represent the Proxy S-PTs which are embedded at each satellite. Each Proxy S-PT is combined with the *Client/Server* Behavioural Pattern, where the “proxy” element in the S-PT is the “client”, and the “subject” is defined with the “server” role. The “subject” at each Proxy is designated as “Remote Executor (RE)”. The flow of information between the nucleus and the proxies is ruled by the *Streaming* Behavioural Pattern. An operator sequence to represent this reconfiguration of the first stage of the pipeline might be :

```
Create( ProxySP, ``proxy1`` )
DefineBehavPatt( proxy1, ``Client/ServerBP`` )
Replicate( 3, proxy1, { ``proxy2``, ``proxy3``, ``proxy4`` } )
Create( StarSP, ``starDP1`` )
DefineBehavPatt( starDP1, ``StreamingBP`` )
Embed( proxy1, starDP1, ``satellite1`` )
Embed( proxy2, starDP1, ``satellite2`` )
Embed( proxy3, starDP1, ``satellite3`` )
Embed( proxy4, starDP1, ``satellite4`` )
```

The elements in this new configuration still have to be instantiated, i.e. the “real subject” in “proxy1” should be associated to a particular “Remote Executor”, etc.

Please note that the Proxy S-PT combined with the *Client/Server* BP may be considered a *Regular* pattern as described in section 5.2.2. This means that it is possible to extend the structure of this pattern where the behaviour of the new element is automatically defined. For instance, in case the data at the “RemoteExecutor” in “proxy3” becomes inaccessible for some reason, it is possible to forward the access to a replicated (backup) repository.

Figure 7.32 shows such a possible reconfiguration scenario in the satellite containing the “proxy3”. The application of the *Extend(proxy3)* operator results on the creation of the element “ProxyB” which is aimed to forward to the (new) “Remote Executor (RE)” the incoming requests generated by the “Proxy” element. The “ProxyB” is automatically defined as a “server” to the pre-existent “Proxy” and also as a “client” of the “RE” (i.e. the “server”).

Whereas the previous scenario concerns a development time reconfiguration, it is also possible to define an hypothetical dynamic reconfiguration scenario concerning the “farmStar”

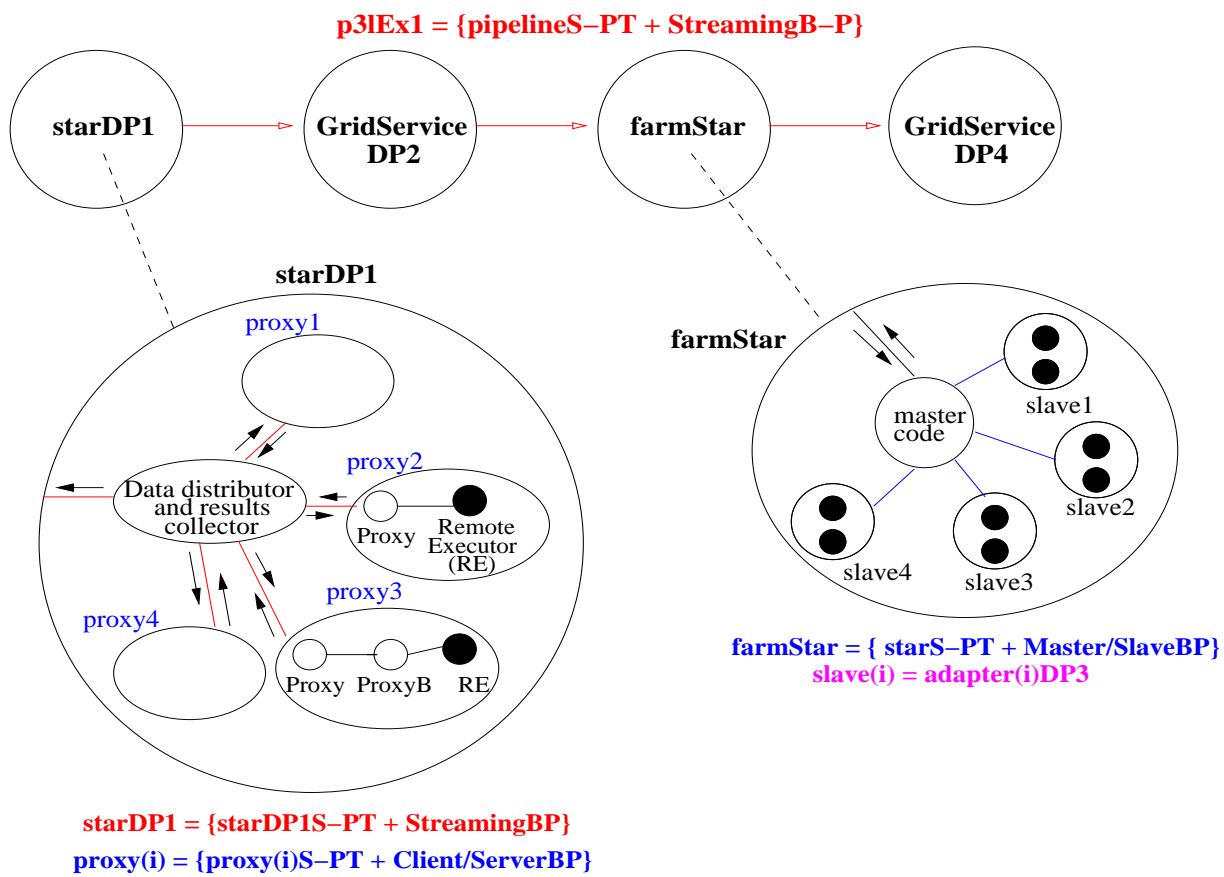


Figure 7.32: Two reconfiguration scenarios for the modelling presented in Figure 7.31.

pattern in the third stage of the pipeline. This reconfiguration scenario is also presented in Figure 7.32. First of all, since the “farmStar” is also a *Regular* pattern resulting from the combination of a Star S-PT and the *Master/Slave* Behavioural Pattern, new added satellites will be automatically defined as “slaves” within that BP. We recall that the dynamic reconfiguration of a *Regular* pattern was discussed throughout section 5.3.

A dynamic reconfiguration of the “farmStar” pattern might be useful in case of need to speed up the processing of the income data at that “farmStar” pattern. Therefore, a new satellite (“slave4” in the Figure) may be created to support the execution of another replica of DP3. Such reconfiguration is possible at execution time because the pre-existing slaves are not disturbed, but of course the “master code” must acknowledge at run-time the existence of new replicas of DP3 (i.e. slaves) to which send the incoming data. This reconfiguration scenario might be supported by the following operator sequence:

```
Replicate( 1, p3lEx1.farmStar.adapter3DP3, ``adapter4DP3`` )
Increase( 1, p3lEx1.farmStar )
Embed( adapter4DP3, p3lEx1.farmStar, ``slave4`` )
```

As described in 4.3.3 which discusses the structural operation of Hierarchical Pattern Templates, the replica generated by the *Replicate* operator is created at the same level of the (out-most) operated Hierarchical Pattern. Namely, “adapter4DP3” is created outside the “p3lEx1” pattern, and therefore has to be included at the right place through the *Embed* operator.

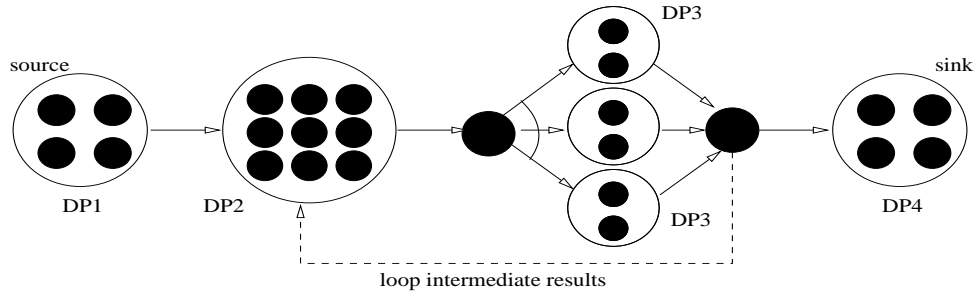


Figure 7.33: A P3L pipeline with two iterated stages. Results of stage 3 are fed back to stage 2 [151].

The second reconfiguration example is based on a modification of the P3L example previously presented in Figure 7.29. Specifically, a *loop* Control Parallel skeleton (CP) is added to the configuration in order to execute more than once the skeletons controlled by the loop CP. Figure 7.33 shows such a case, where the results of the third pipeline stage are fed back to the second stage, according to the loop policy.

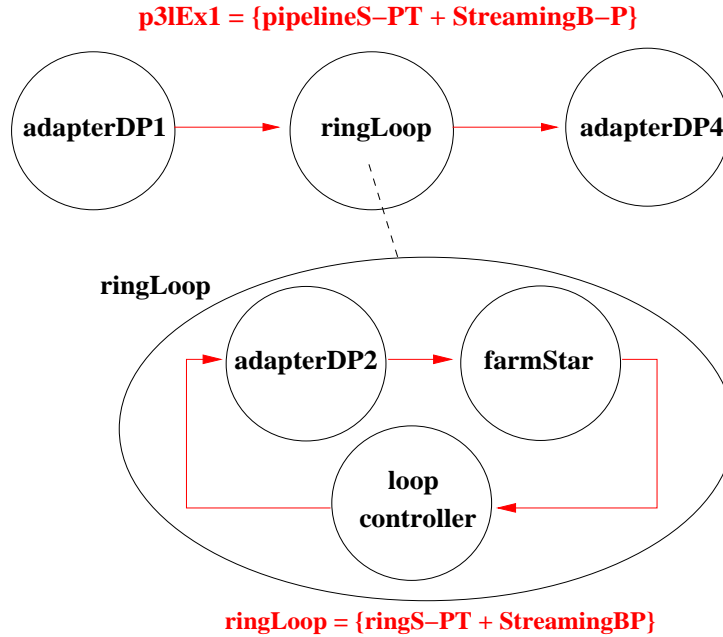


Figure 7.34: Modelling the P3L example in Figure 7.33 using the Pattern/Operator model.

To represent this second reconfiguration case, the pattern-based modelling formerly presented in Figure 7.30 is modified as depicted in Figure 7.34. Please note that such modification could also have been done to the pattern-based modelling presented in Figure 7.31.

The Ring Structural Pattern combined with the *Streaming* Behavioural Pattern in Figure 7.34 is used in order to model the loop skeleton, as described earlier in section 7.4.1. This “ringLoop” pattern includes the “adapterDP2” and “farmStar” elements, which formerly were the second and third stages of the pipeline, and the “loop controller” element that enforces a condition-based policy. The “p3LEx1” has now only three stages, and the “ringLoop” is embedded in the

second stage. A possible operator sequence to reconfigure, at development time, the modelling in Figure 7.30 into the configuration in Figure 7.34 may be, for example:

```
Terminate( p3lEx1 )
Extract( adapterDP2, p3lEx1, ``cph2`` )
Extract( farmStar, p3lEx1, ``cph3`` )
Decrease( 1, p3lEx1, ``cph3`` )
Create( RingSP, ``ringLoop``, 3 )
DefineBehavPatt( ringLoop, ``Streaming`` )
Embed( adapterDP2, ringLoop, ``cph2`` )
Embed( farmStar, ringLoop, ``cph3`` )
Instantiate( ringLoop, ``cph1``, ``loop controller`` )
Embed( ringLoop, p3lEx1, ``cph2`` )
Start( p3lEx1 )
```

The names “cph1”.. “cph4” in the script identify the stages in the original pipeline, and we recall that the extracted patterns are moved to the same level of the outmost pattern, i.e. “p3lEx1”.

Please note that in case it is possible to make the above reconfiguration without having to abort the execution of the overall elements, the *Stop* and *Resume* operators could be used in the above script instead of the *Terminate* and *Start* operators, respectively. Through the *Stop* operator the execution of all patterns would be suspended and a checkpoint of the execution state is made. It is assumed that the *Stop* operator is applied recursively to all embedded patterns. With the the *Resume* operator the saved checkpoint state is restored and the execution is resumed.

These two ways of reconfiguring an application were previously discussed throughout section 5.3, namely: a) to abort an application’s execution, apply the necessary modifications, and re-execute it; and b) to suspend the application’s execution, completely or partially, and apply the desired changes.

7.5 Analysis of Gravitational Waves

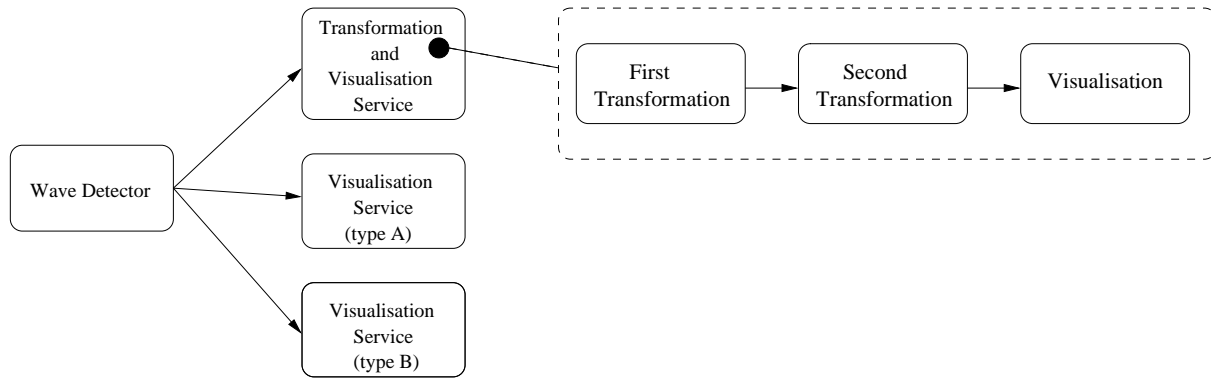


Figure 7.35: A simple example in the area of gravitational wave experiments.

Astrophysics is one of the areas that can also benefit from the computational power and the distributed nature of Grid computing. The applications in this area require high-performance computing support for the scientific calculations, which generate huge amounts of data. Additionally, the involved scientists and scientific instruments belong to different organisations that combine efforts to solve applications' intrinsic complex problems. One particular area is related to gravitational wave experiments [75] where out of space waves are detected and analysed. Figure 7.35 shows a simple example where a wave detector is producing data to be analysed and displayed by several services, allowing scientists to compare the results. Two of those services are represented in the example by two different types of visualisation services. The third service applies a sequence of transformations to the original signal and displays the results.

To configure this application example, the user first identifies the relevant Structural Patterns. A star topological PT with three *satellites* is created to represent the connections between the *Wave Detector* service (Figure 7.35) and the transformation and visualisation services. In turn, to support the *Transformation and Visualisation service* (Figure 7.35) the user creates a pipeline topological PT also with three elements. To obtain the right number of component place holders in both pattern templates, the user possibly had to apply the *Increase()* or the *Decrease()* operators (e.g. if the created star PT does not have enough satellites by default). Subsequently, the user combines both pattern templates by embedding the pipeline PT into one of the star's satellites. Finally, the user instantiates the component place holders with the adequate services.

7.5.1 Simulation in Triana

This section presents a simplified implementation of the above example, which was actually built on our prototype, due to the absence of an available tool in Triana for gravitational wave detection. Namely, the detector is represented by a component which generates a wave with parameterisable amplitude, frequency, etc.

The first configuration step is the creation of the two required PTs: a Star PT represents the connections between the *Wave Detector* service (Figure 7.35) and the transformation and visual-

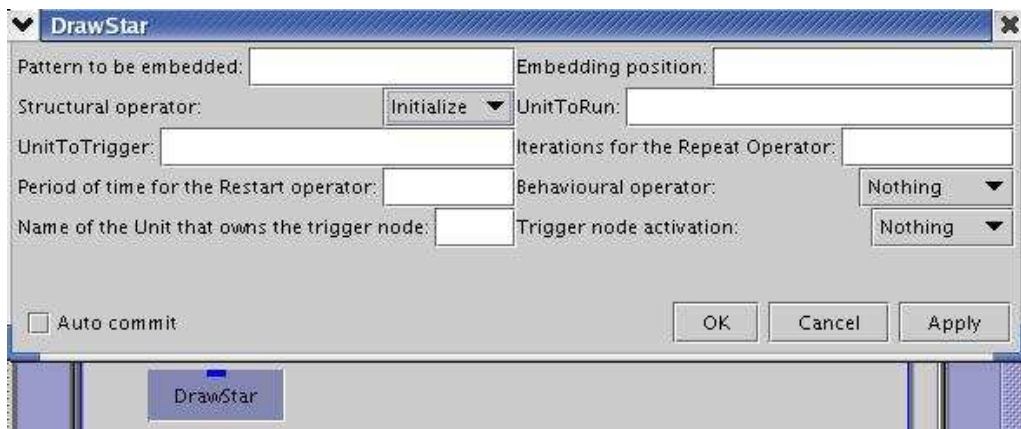


Figure 7.36: *Initialisation of a Star PT.*

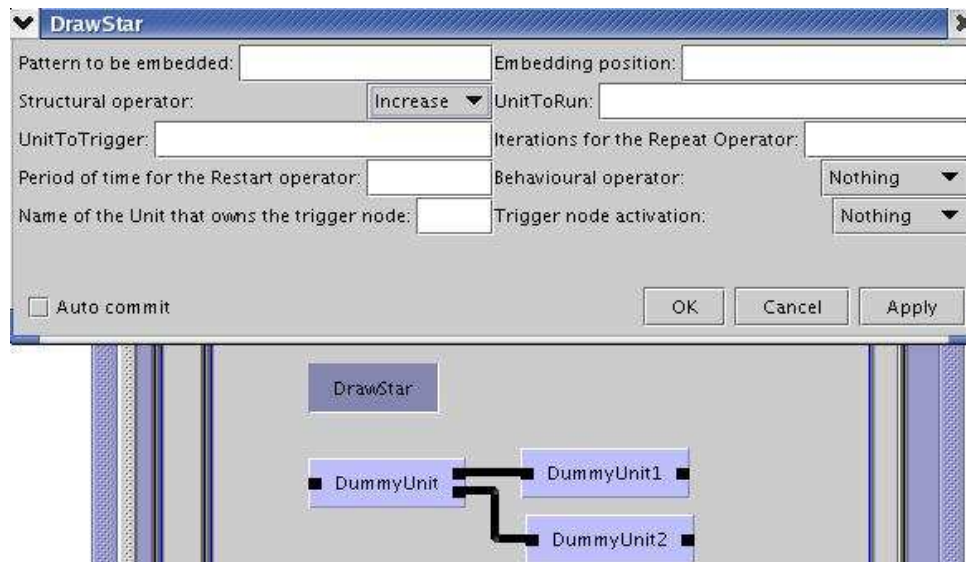


Figure 7.37: *Addition of one satellite to a Star PT.*

isation services; a Pipeline PT supports the *Transformation and Visualisation service* (Figure 7.35). In order to create a Star PT, the user drags and drops the *DrawStar* unit from Triana's Patterns toolbox and initialises it (Figure 7.36). Depending on the number of satellites created by default, the user may have to apply the *Increase()* or the *Decrease()* operators to the Star PT. In this case, it is necessary to increase the number of satellites (Figure 7.37). For the creation of the Pipeline PT, the user selects the *DrawPipeline* unit and repeats the process.

Figure 7.38 shows the two PTs already including the right number of component place holders. These place holders are represented by *DummyUnit* components that can be instantiated to a Structural Pattern Template, or to a service (unit) from the toolbox.

The next configuration step is to structure the two PT templates so that the *Transformation and Visualisation service* is connected to the *Wave Detector* service. As such, the user applies the *Embed* operator to the Star PT to transform the Pipeline PT into one of the star's satellites (*DummyUnit1* in Figure 7.39).

Finally, the user instantiates the pattern templates with the necessary services from the toolbox. An example can be seen in Figure 7.40.

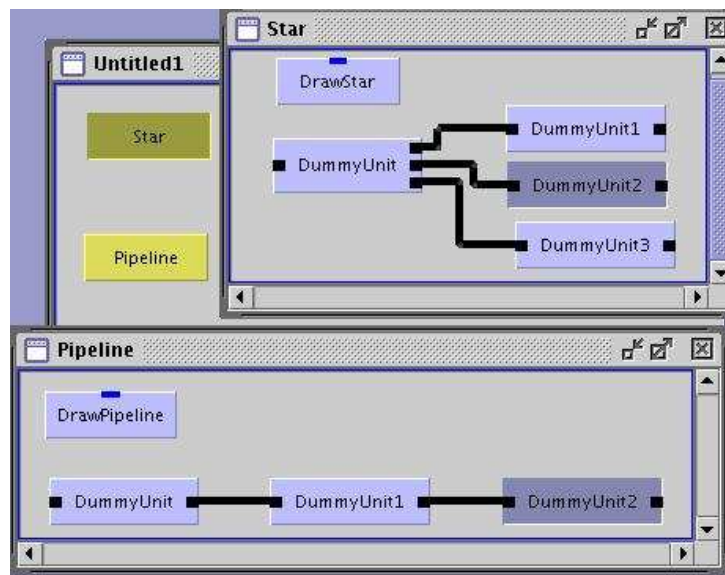


Figure 7.38: A Star PT with three satellites and a Pipeline PT with three elements.

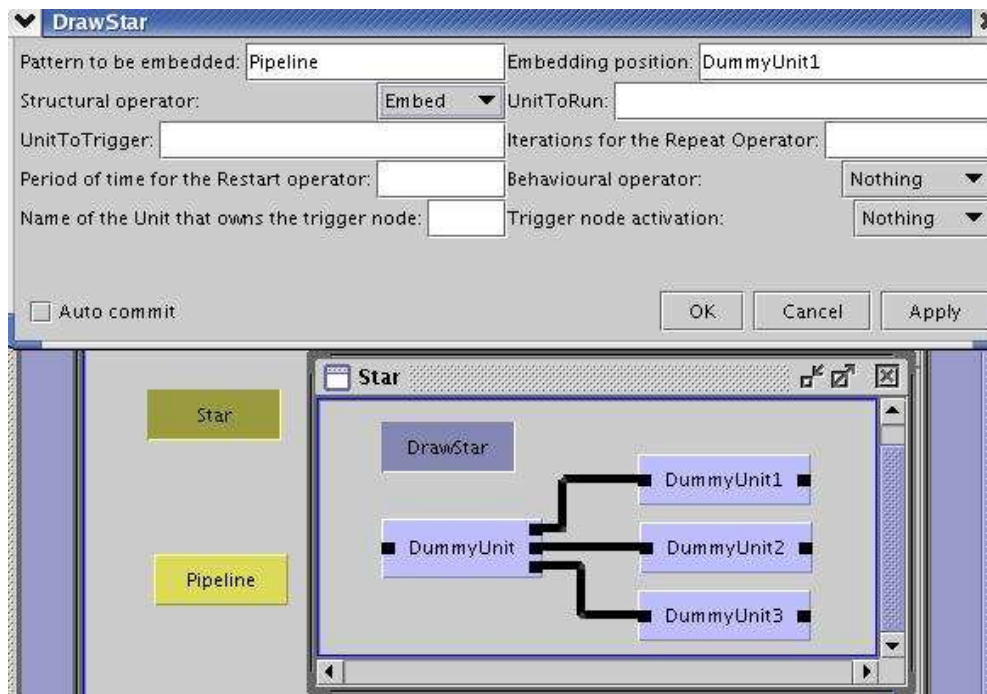


Figure 7.39: Application of the Embed Structural Operator to the Star PT.

Figure 7.41 shows the final configuration after all the template slots have been instantiated. For demonstration purposes, as said earlier, the *Wave detector* (Figure 7.35) is represented in this example by the *Wave* unit which generates a waveform (the users may configure parameters like frequency, amplitude, type of wave, etc). Two graphical displaying units for rendering input signals are selected to represent the visualisation services: the *SGTGrapher* and the *Histogrammer*. The selected transformation services for instantiating the first two pipeline stages are the *Gaussian* unit (which adds noise to the data generated by the *Wave*) and the *FFT* unit (which performs a Fast Fourier transform).

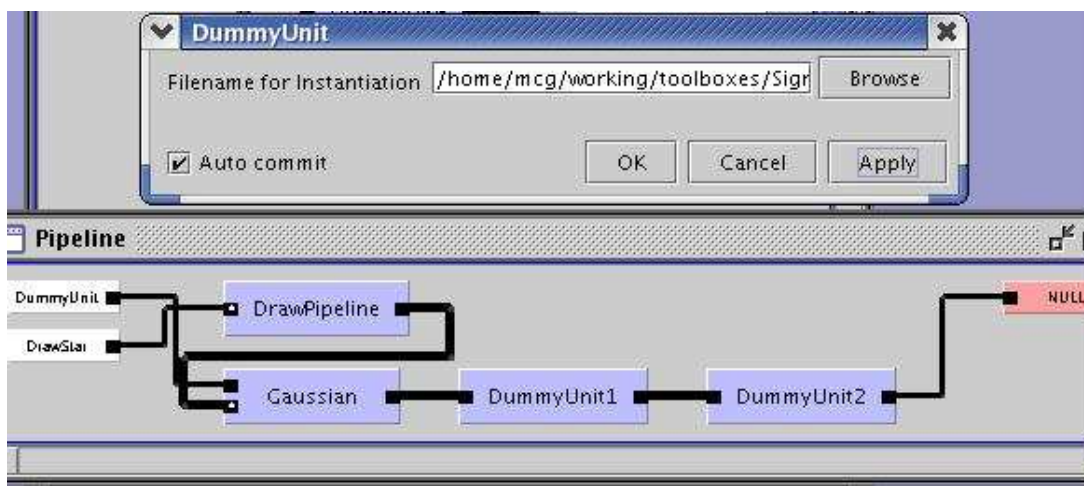


Figure 7.40: *Instantiation of a Unit.*

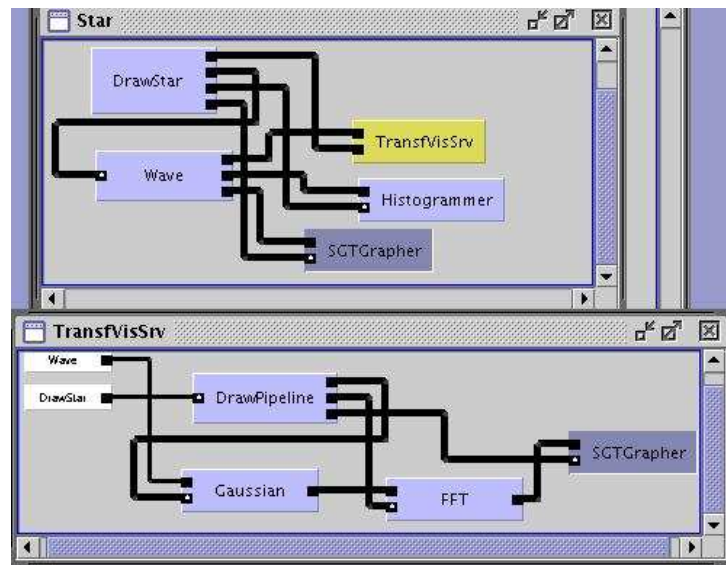


Figure 7.41: *Final configuration.*

After the execution, the user may compare the same data without transformations using two different displaying units. It is also possible to compare that data to the transformed data in a third display unit(Figure 7.42).

7.5.2 Configuration and Execution through a Script

The simulation described in section 7.5.1 can be automated through a script. Considering the original example where a wave detector is constantly producing data, it might be interesting to restart the application periodically. The script of the simulation, as described ahead, includes the Restart Behavioural operator as the last operation, and launches the execution every 20000 milliseconds. The restarting can be aborted at any time by calling the Terminate Behavioural operator.

Figure 7.43 shows the application of the script for the Wave detection simulation, and its code is as follows (the lines are numbered for its reference within the text):

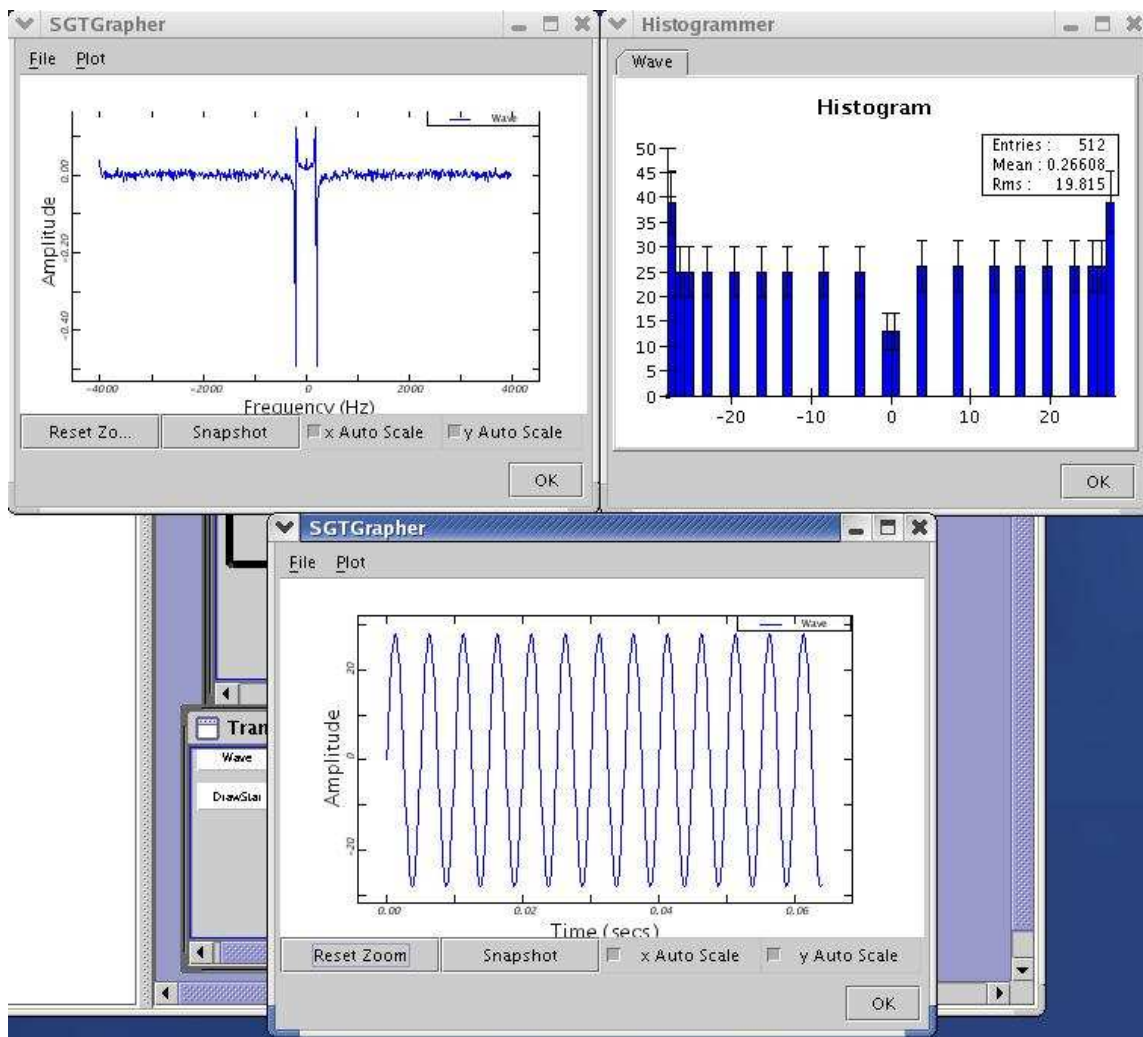


Figure 7.42: *Execution results.*

```

1: Initialize
2: Increase 1
3: Create Pipeline TransfVisSrv
4: RunStructuralScript TransfVisSrv
5: Instantiate DummyUnit
    /home/mcg/working/toolboxes/SignalProc/Injection/Gaussian.xml
5: Instantiate DummyUnit1
    /home/mcg/working/toolboxes/SignalProc/Algorithms/FFT.xml
6: Instantiate DummyUnit2
    /home/mcg/working/toolboxes/SignalProc/Output/SGTGrapher.xml
7: EndStructuralScript
8: Embed TransfVisSrv DummyUnit1
9: Instantiate DummyUnit
    /home/mcg/working/toolboxes/SignalProc/Input/Wave.xml
10: Instantiate DummyUnit2
    /home/mcg/working/toolboxes/SignalProc/Output/Histogrammer.xml

```

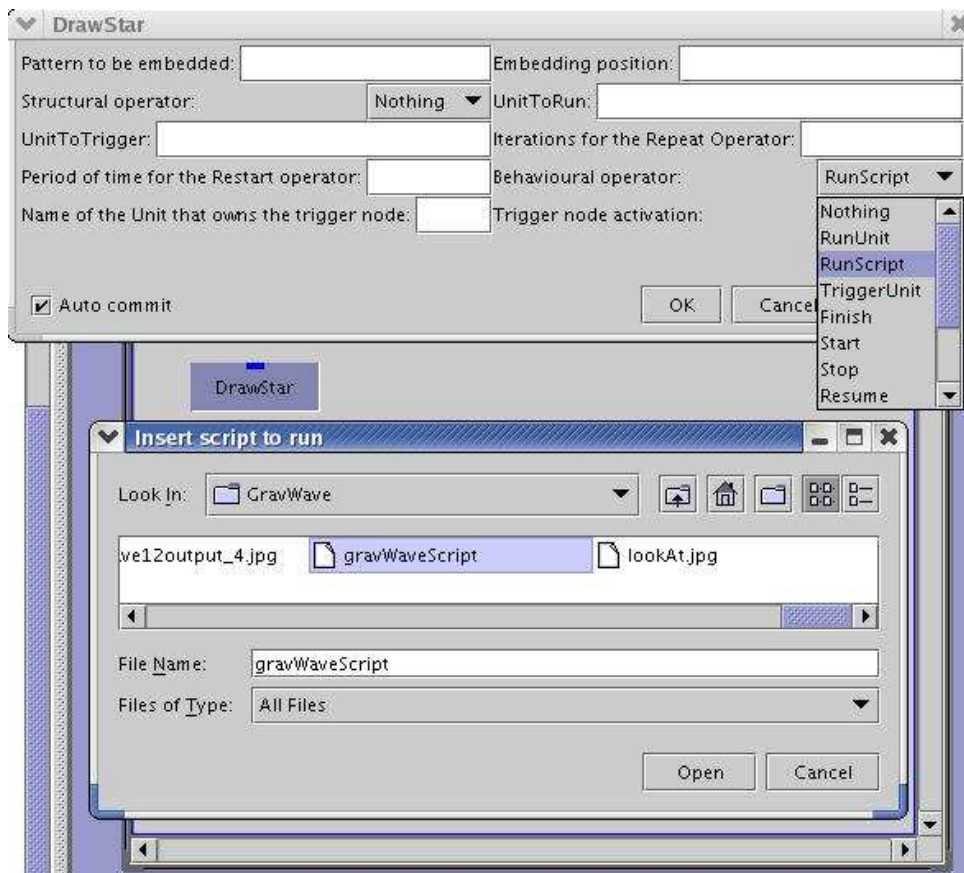


Figure 7.43: Application of a script to a pattern template.

```

11:Instantiate DummyUnit3
    /home/mcg/working/toolboxes/SignalProc/Output/SGTGrapher.xml
12:Restart 20000

```

The script is run by the pattern controller of a Star PT which performs the following steps: a) creates the Star (1); b) adds one satellite to the nucleus (2); c) creates a Pipeline PT (named *TransfVisSrv* – line 3) and instantiates all its slots (called *DummyUnit(i)* – lines 4-7); d) embeds the Transformation and Visualisation service (*TransfVisSrv*) into the first satellite (*DummyUnit1* – line 8); e) instantiates the rest of the empty slots of the template (lines 9-11); and, applies the *Restart* Behavioural Operator, in order to execute the instantiated Star every 20000 milliseconds (12).

In Figure 7.44 the debug window displays auxiliary messages in the code including the re-activation of the execution through the *Restart* Behavioural Operator.

7.5.3 Simulating Regular Production of Data

The simulation of the analysis of gravitational waves example described in sub-section 7.5.1 does not take into account the regular production of data by the wave detector. To simulate such situation, this section describes a very simple configuration (Figure 7.45) where the tool *Count* produces different values, at each execution, to the frequency parameter of the Wave tool. Frequency starts at value 100Hz, and it is increased at each execution by 100Hz until a

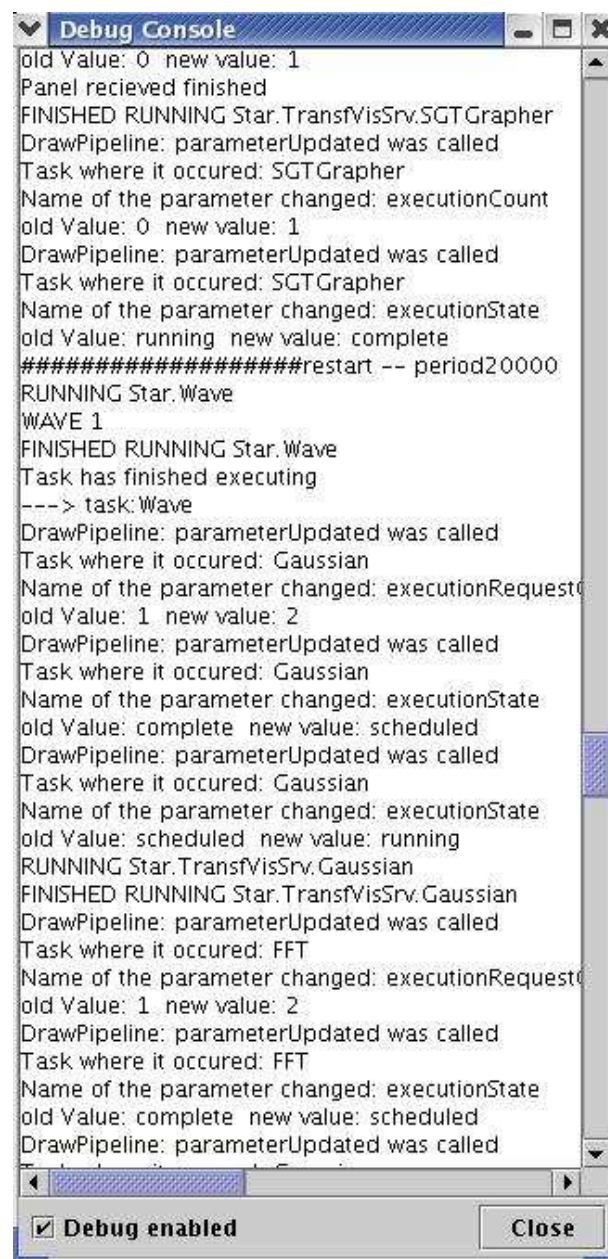


Figure 7.44: Debug window showing the application of the Restart Behavioural operator.

maximum of 4000Hz. Consequently, the Wave tool produces different waves which can be visualised in the *SGTGrapher* tool.

Similarly to the previous section 7.5.2, the *Restart* Behavioural operator can be applied to the simulation, to see a sequence of different waves at a fixed time period (10 seconds). Figure 7.47 shows two consecutive snapshots of the *SGTGrapher* tool.

The automatic re-execution can be stopped at any time by applying the *Terminate* Behavioural operator (Figures 7.48 and 7.49). The *Count* tool remembers the intermediate value for the frequency parameter of the last execution. Therefore, the user may, for example, repeat the execution a certain number of times, by restarting from the previous saved frequency value.

Figure 7.50 shows the selection of the *Repeat* Behavioural operator for repeatedly launching the execution of the simulation, by a certain number of times (in this case, 10 times). In this way, the user can see the result after each consecutive iterations. The debug window in the

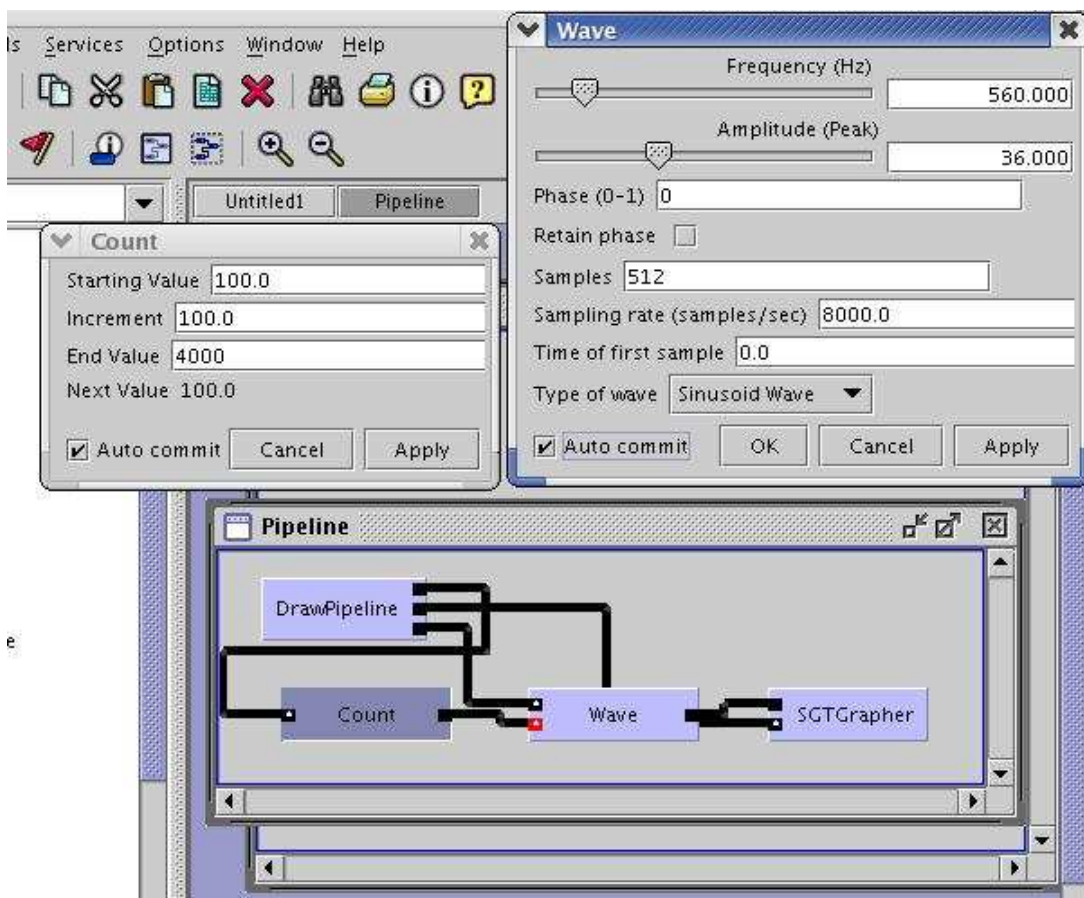


Figure 7.45: A simple simulation of regular production of data by the gravitational wave detection service.

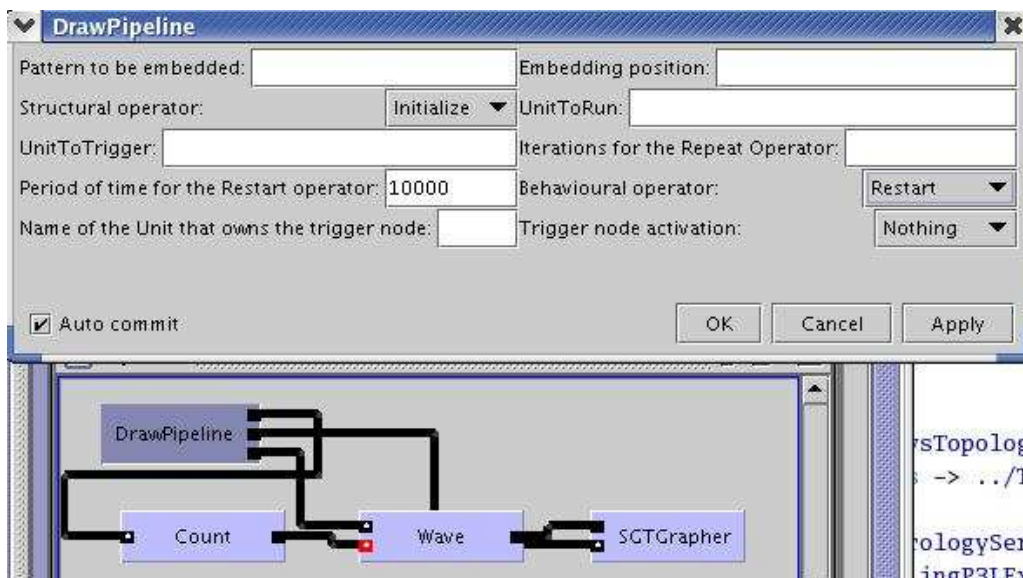


Figure 7.46: Producing different waves every 10 seconds.

Figure shows that the *Repeat* operator was repeatedly called.

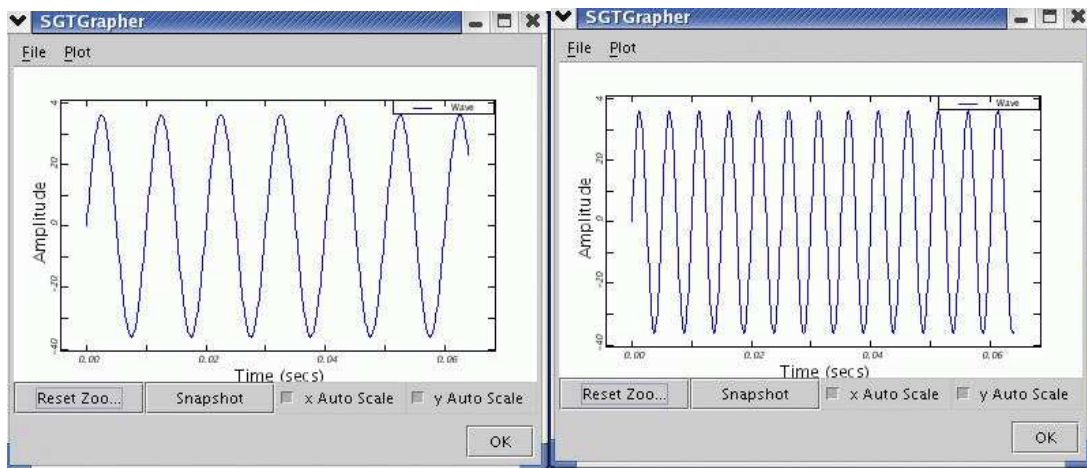


Figure 7.47: Two different waves produced at two consecutive execution steps.

Figure 7.48: Selection of the Terminate Behavioural operator.

```

RUNNING Pipeline.Count
FINISHED RUNNING Pipeline.Count
Task has finished executing
---> task:Count
Parameter that was updated:behaviouralOperator value:Terminate
Inside terminate ++++++
DrawPipeline: parameterUpdated was called
Task where it occurred: DrawPipeline
Name of the parameter changed: behaviouralOperator
old Value: Nothing new value: Terminate
#####restart stopped

```

☒ Debug enabled

Close

Figure 7.49: The debug window showing the execution of the Terminate Behavioural operator.

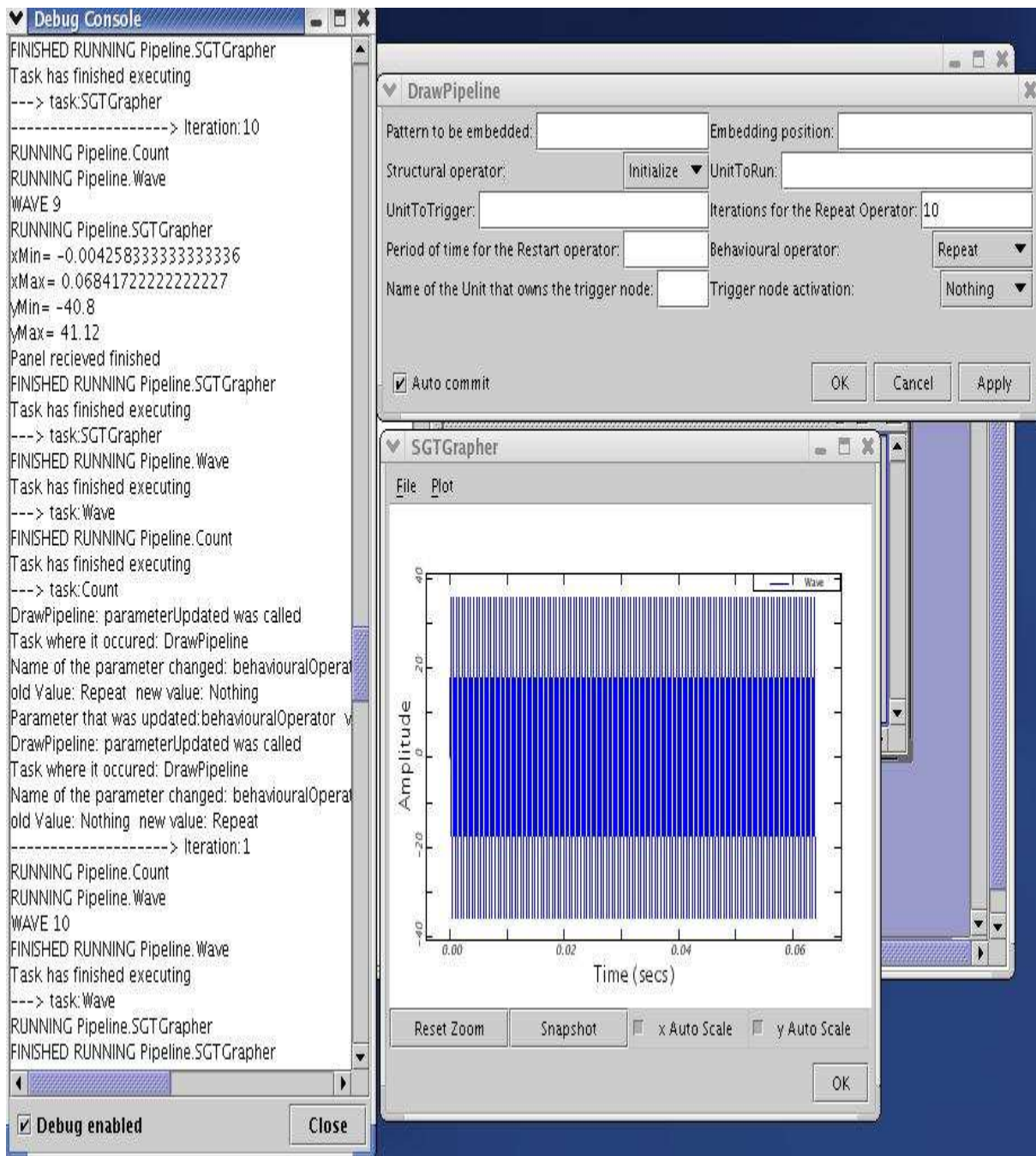


Figure 7.50: Applying the Repeat Behavioural Pattern for launching the execution ten consecutive times.

7.6 Galaxy Formation Example

To further illustrate the use of Pattern Templates in Triana, we use a “Galaxy Formation” code example. The present example is an extension of the version introduced in [44].

The “Galaxy Formation” example involves generating the position of particles and subsequently animating these – using a combination of “DataReader” and “Animation” modules from Triana. A data file is loaded by a single Data Reader Unit within Triana, and passed to all the Triana nodes. Nodes then buffer the data for future calculations. Note that the data file could be copied beforehand and distributed in a parallel way also. The loaded data is then separated into frames, distributed amongst the various Triana servers on the available network and processed to calculate the column density using smooth particle hydrodynamics. These types of simulations can usually generate large data files containing snapshots of an evolving system. They are therefore quite representative of the types of applications that may be executed over a Grid infrastructure. In this particular example, after undertaking a simulation run, a snapshot is produced – and which is independent of others over time. This suggests that any data analysis on frames can be carried out independently. Grid resources are used in this instance to distribute and remotely process data frames, which finally return a small image to the visualisation/controlling client. The images can be subsequently re-assembled in real-time into the correct chronological order to generate a smooth animation.

Galaxy and star formation simulation codes generate binary data files that represent a series of particles, along with their associated properties as a snapshot in time. The user of such codes would like to visualise this data as an animation in two dimensions, with the ability to vary the perspective of view, and project that particular two dimensional slice and re-run the animation. Due to the nature of the data, each frame or snapshot is a representation at a particular point in time of the total dataset. It is possible to distribute each time slice or frame over a number of processes and calculate the different views based on the point of view in parallel.

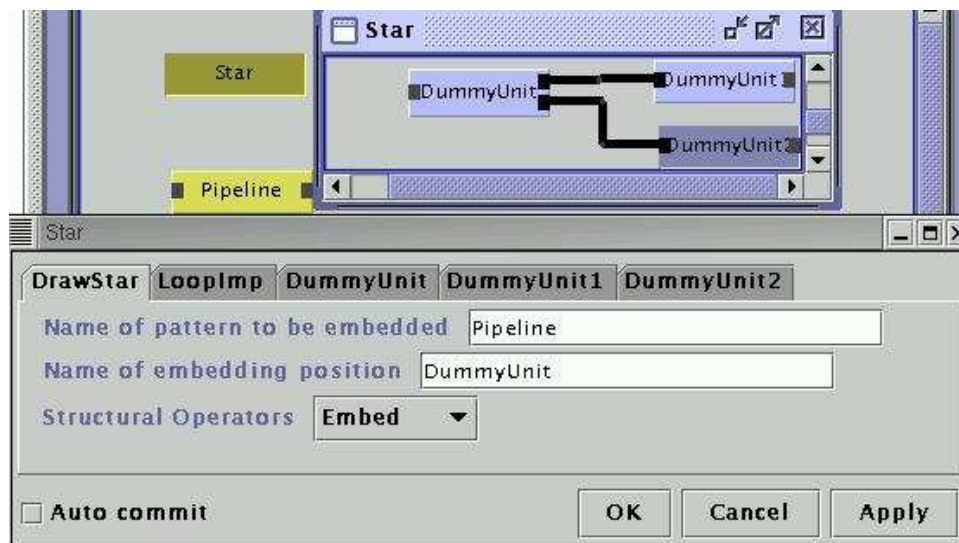


Figure 7.51: The animation is supported by a pipeline PT which is embedded in the nucleus of the star PT.

The Galaxy formation example may be represented by a Star PT whose nucleus contains the

actions necessary to generate and control the animation execution, and the satellites represent image processing and analysis actions. In this way, the same animation can be simultaneously analysed/processed in different ways. Figure 7.51 shows a Star PT with three component place holders – the satellites (*DummyUnit1* and *DummyUnit2*) and the nucleus (*DummyUnit*). As the animation is developed in stages, these are represented by a Pipeline PT. Figure 7.51 shows the Pipeline PT embedded in the nucleus of the star by selecting the *Embed Structural Operator*, and by identifying the embedding position (*DummyUnit*).

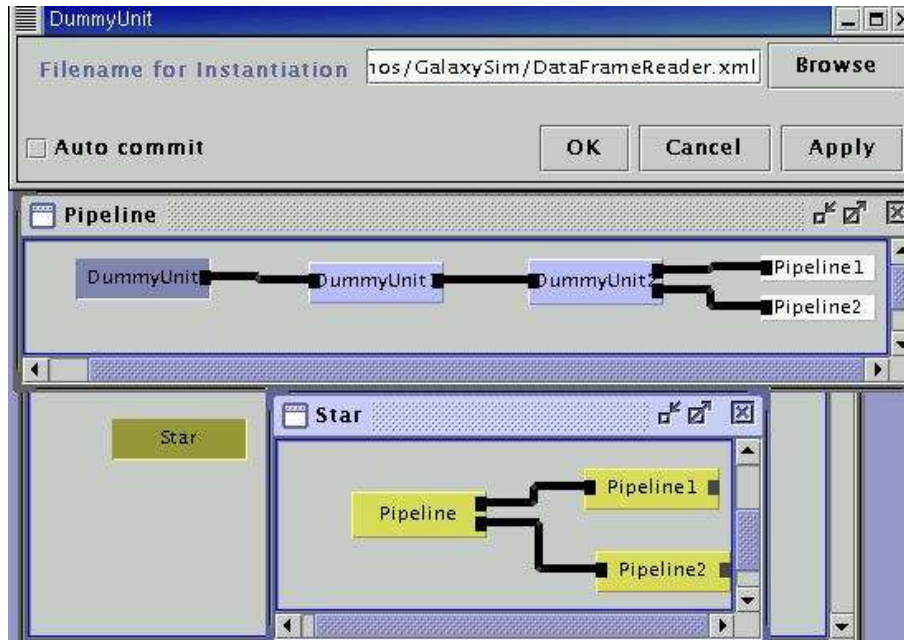


Figure 7.52: An example of a component place holder instantiation.

Figure 7.52 shows the Star PT with the embedded Pipeline PTs, to support the image processing activities required to generate the animation. The snapshot represented by the Figure was taken with the first implementation of patterns and operators over Triana where only the Structural Operators were available. The next step involves instantiating the place holder (named *DummyUnit*) of the pipeline (in this case a *DataFrameReader* is selected from the Triana toolbox) – as illustrated in Figure 7.52. Figure 7.53 shows the final configuration, with all component place holders instantiated with units. Hence, the binary data file produced by the simulation code is loaded by the *DataFrameReader* unit. The frames are sent to the *SequenceBuffer* unit – a media controller that allows the replay of the application. The user may stop the animation, rewind it, restart it, etc. The *ViewPointProjection* unit takes the 3D data and maps this onto a 2D space outputting a standard *PixelMap*. The user may change the point of projection by changing parameters representing the (x,y) coordinates. The resulting animation images are analysed/processed in parallel in *Pipeline1* and *Pipeline2*. The *GradientEdge* unit selects images based on a gradient edge detector, and subsequently displays these using the *ImageView* unit. In *Pipeline2*, the number of non-black objects in each image are counted by *CountBlobs* unit and displayed in *ConstView* unit.

Figure 7.54 shows the output of units *ImageView* and *ConstView*, and shows the parameter interface panel for unit *SequenceBuffer*.

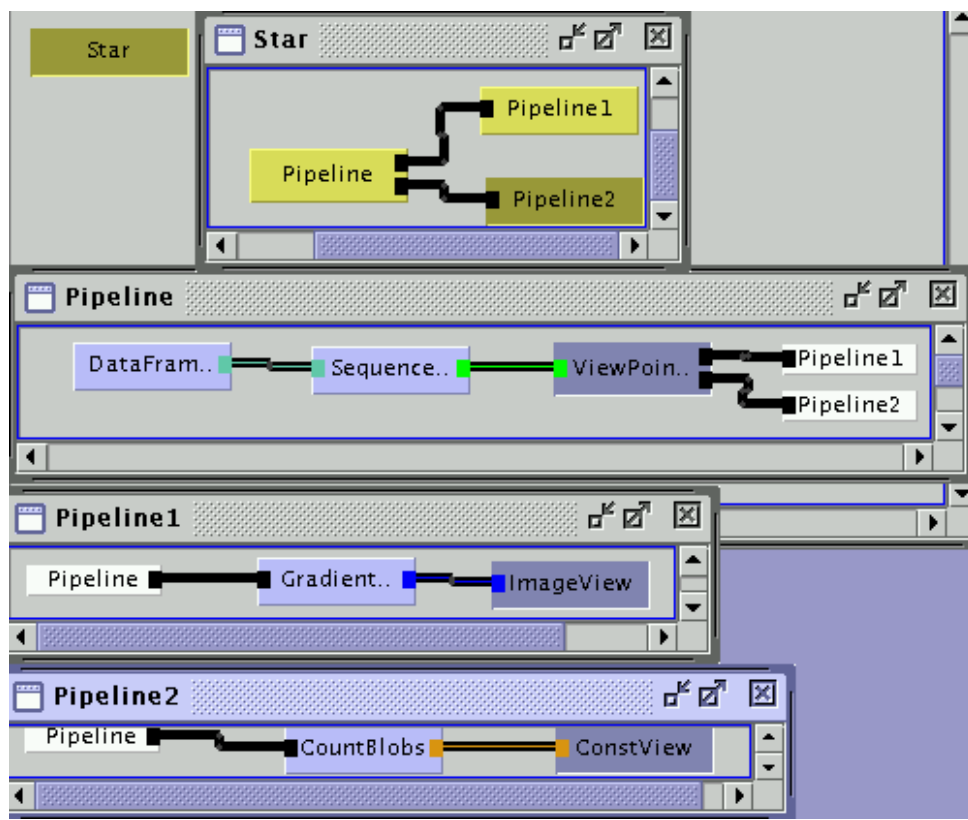


Figure 7.53: A possible final configuration for the image processing of the “Galaxy Formation example”.

7.6.1 Alternative Configuration

A possible alternative configuration decouples the viewpoint projection of the simulation from the reading of data, allowing parallel animations with different viewpoints. A Star PT supports the configuration (Figure 7.55): the data is read at the nucleus by the *DataFrameReader* unit (the *SequenceBuffer* unit was omitted for simplification) and sent to the satellites *Pipeline* PT and *Ring* PT to be processed. The *DataFrameReader* unit may interact with the satellites according to a *Streaming* Behavioural Pattern. In the satellite supporting the *Pipeline* PT (see Figure 7.55), a user may select the appropriate viewpoint through the *ViewPointProjection* unit. The resulting images may be scaled by the *ScaleImage* unit and subsequently displayed by the *ImageView* unit. The *Producer/Consumer* Behavioural Pattern may represent the interaction between the *ScaleImage* (the producer) and the *ImageView* unit.

In the satellite with the *Ring* PT (Figure 7.55), the viewpoint is automatically selected according to the number of non-black objects in each image. For the *Pipeline1* stage contained within the *Ring* PT (see Figure 7.55), the images produced by *ViewPointProjection* are visualised in the *ImageView* unit. In the next stage of the ring, the *CountBlobs* unit counts the number of non-black objects in each image, followed by a stage (*Pipeline*) which evaluates if it is necessary to change the viewpoint. If this is the case, the *Scroller* unit is triggered and inputs the new value to the “x” coordinate parameter for the unit *ViewPointProjection*, thereby closing the ring.

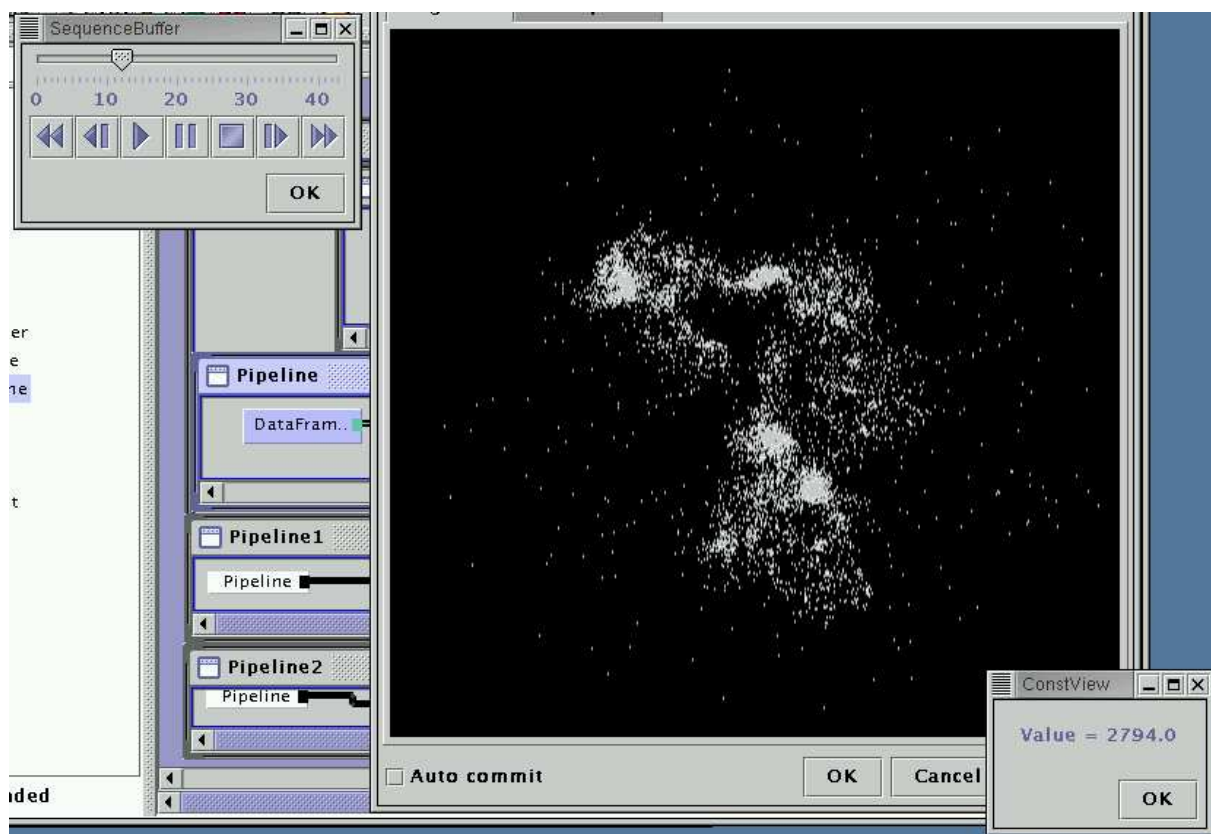


Figure 7.54: Execution snapshot for the selected configuration.

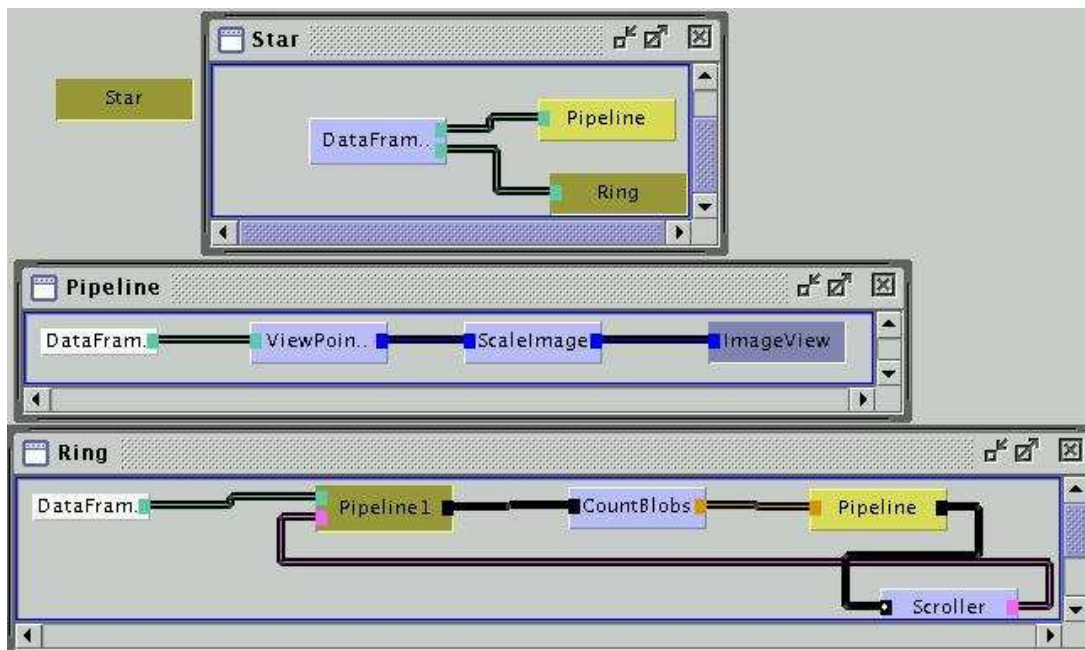


Figure 7.55: Parallel animation execution with different view points.

7.6.2 Introducing Execution Control and Reconfiguration

To allow a step-by-step execution of the Galaxy simulation example, our implementation relies on *trigger nodes* provided by Triana (see section 6.4.3). Each *pattern controller*, associated with

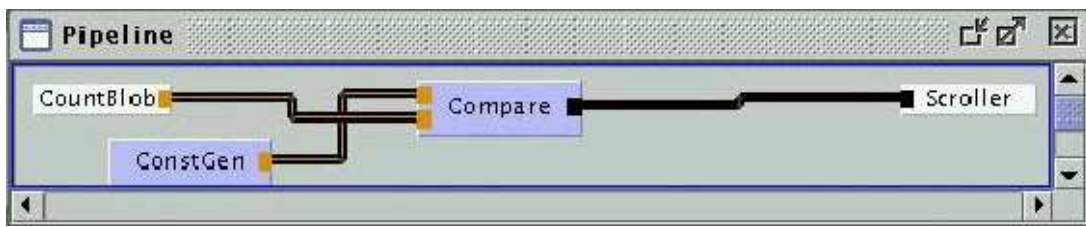


Figure 7.56: Detail of the stage named Pipeline in the Ring PT from Figure 7.55

each Structural Pattern, is connected to all the place holders in the pattern through trigger nodes. The trigger nodes may be in two states: silent, or active. In the silent state, the trigger node has no influence upon the execution, and the control is driven by the data flow. In the active state, execution control is stopped at the unit that owns that trigger node. Although data may arrive in that unit's data nodes, only when the trigger node is "triggered", execution flow is allowed to proceed. The user can, at any time, activate and deactivate a trigger node through the operator panel of the associated pattern controller.

In this way, it is possible to control the execution of a single unit within the pattern. The implementation of the Behavioural Operator *Stop* over a single tool might be implemented in this way – to stop a unit would simply imply that its pattern controller would activate the unit's trigger node. The *Resume* Behavioural Operator, limited to the next iteration, would simply require triggering the unit. To continue the execution until the end (complete *Resume*) it would simply require the deactivation of the trigger node.

Back to the example, to simplify the instantiation and composition of patterns, the configuration and start of execution of a similar version of the Galaxy simulation example are defined in a script.

Figure 7.57 shows how to start the interpretation of the script *GalaxyExecCtrl*. The unit that represents a Star PT has to be dragged into the scratch-pad and its execution launched. The user selects the *RunScript* operation and the intended script.

The script, containing the Structural and Behavioural Operators, is as following (lines are numbered for their reference within the text):

```

1: Initialize
2: Create Pipeline ImgProjection
3: RunStructuralScript ImgProjection
4:   Decrease 1
5:   Instantiate DummyUnit
        /home/mcg/working/toolboxes/Demos/GalaxySim/DataFrameReader.xml
6:   SetParameter DataFrameReader fileName /home/mcg/working/triana/old_out.drt
7:   Instantiate DummyUnit1
        /home/mcg/working/toolboxes/Demos/GalaxySim/ViewPointProjection.xml
8: EndStructuralScript
9: Embed ImgProjection DummyUnit
10: Create Pipeline ImgProcessing
11: RunStructuralScript ImgProcessing

```

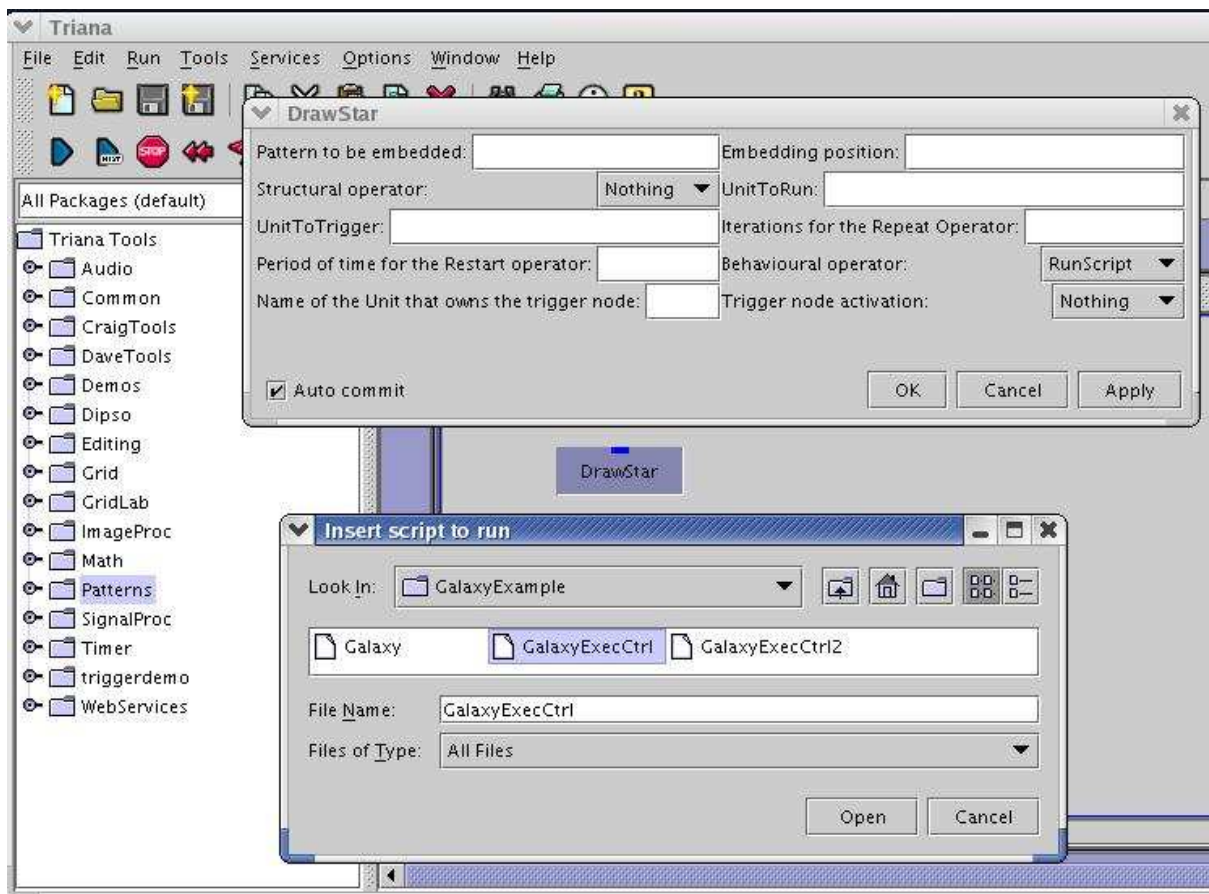


Figure 7.57: Configuration and execution of the Galaxy simulation example through the script named "GalaxyExecCtrl".

```

12: Decrease 1
13: Instantiate DummyUnit
    /home/mcg/working/toolboxes/ImageProc/Processing/Effects/EnhContrast.xml
14: Instantiate DummyUnit1 /home/mcg/working/toolboxes/ImageProc/Output/ImageView.xml
15: Activate EnhContrast
16:EndStructuralScript
17:Embed ImgProcessing DummyUnit1
18:Activate ImgProcessing
19:Create Pipeline ImgAnalysis
20:RunStructuralScript ImgAnalysis
21: Decrease 1
22: Instantiate DummyUnit
    /home/mcg/working/toolboxes/ImageProc/Processing/Detection/CountBlobs.xml
23: Instantiate DummyUnit1 /home/mcg/working/toolboxes/Common/Const/ConstView.xml
24: Activate CountBlobs
25:EndStructuralScript
26:Embed ImgAnalysis DummyUnit2
27:Activate ImgAnalysis
28:TriggerUnit ImgProcessing

```

```
29:TriggerUnit ImgAnalysis
30:Start
```

The *Initialize* operation (line 1:) creates a star with two satellites. Next, the operation *Create Pipeline ImgProjection* (2:) creates a pipeline that will contain the stages to read the frames from the data file and to define the viewpoint projection. In the subsequent operation *RunStructuralScript ImgProjection* (3:), the pattern controller of the pipeline is activated and execution control is passed to it. In this way, the Pipeline PT *ImgProjection*'s pattern controller can continue processing the main script to define the adequate number of component place holders and their instantiation. Since the default number of stages in a Pipeline PT is three, the *Decrease Structural Operator* is applied (4:) eliminating one stage. Next, the first component place holder (*DummyUnit*) is instantiated with *DataFrameReader* unit (5:), and its parameter *fileName* is defined (6:). The second component place holder (*DummyUnit1*) is then instantiated with the *ViewPointProjection* unit (7:).

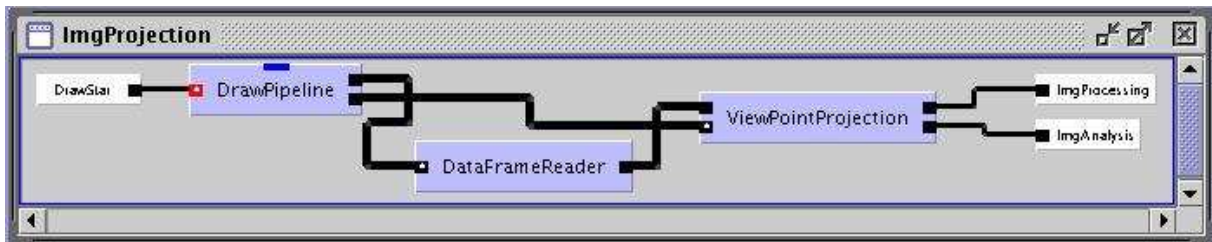


Figure 7.58: The *ImgProjection* pipeline.

The result *ImgProjection* pipeline can be seen in Figure 7.58, where the pattern controller's name is *DrawPipeline*.

The next operation in the main script is *EndStructuralScript* (8:). When the pipeline pattern controller interprets this operation it ends its execution, and the execution control is returned to the Star PT's pattern controller. Next in the main script is the *Embed ImgProjection DummyUnit* Structural Operator (9:) which instantiates the nucleus of the star (*DummyUnit*) with the *ImgProjection* pattern.

The process of creating a pipeline PT is repeated two more times. First, the *ImgProcessing* pipeline PT is created (10:) and embedded in the first satellite (*DummyUnit1*) (17:). *ImgProcessing* contains two stages, one for enhancing the contrast of the 2D image (*EnhContrast* – line 13) produced by the *ViewPointProjection* unit, and another for displaying the image (*ImageView* – line 14).

To control the execution of the pipeline in a step-by-step fashion, the trigger node of the *EnhContrast* was activated with the operation *Activate EnhContrast* (15:). The resulting *ImgProcessing* pipeline can be seen in Figure 7.59.

Second, the *ImgAnalysis* pipeline PT is created (lines 19–25) and embedded in the second satellite (*DummyUnit2*) – line 26. *ImgAnalysis* contains two stages, one for counting the particles (*CountBlobs* counts the non-black objects – line 22) in the image produced by the *ViewPointProjection* unit, and another for displaying that number (*ConstView* – line 23). To control the execution of the pipeline in a step-by-step fashion, the trigger node of the *CountBlobs* was activated with the operation *Activate CountBlobs* (24:).

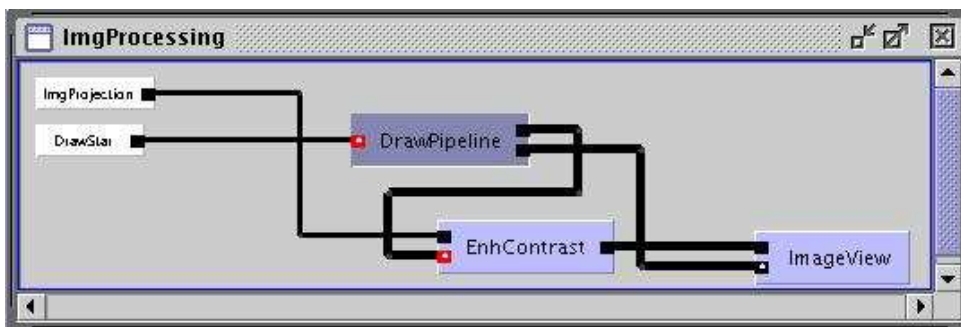


Figure 7.59: The *ImgProcessing* pipeline.

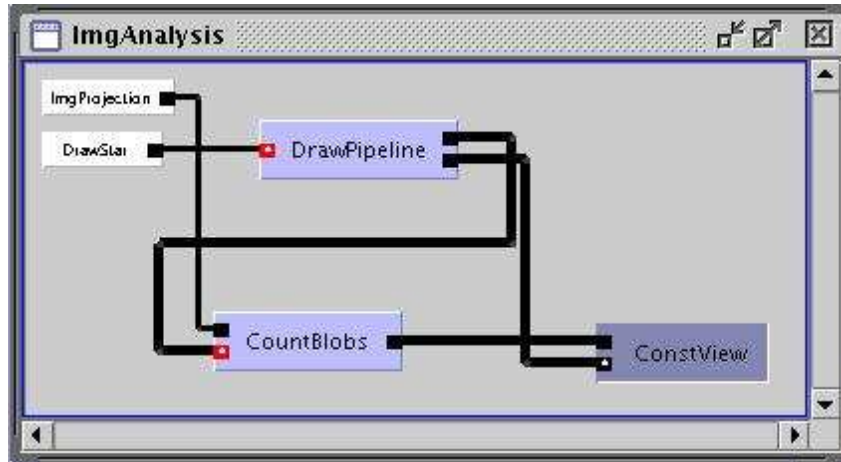


Figure 7.60: The *ImgAnalysis* pipeline.

The *ImgAnalysis* pipeline, after the execution of the entire main script, can be seen in Figure 7.60.

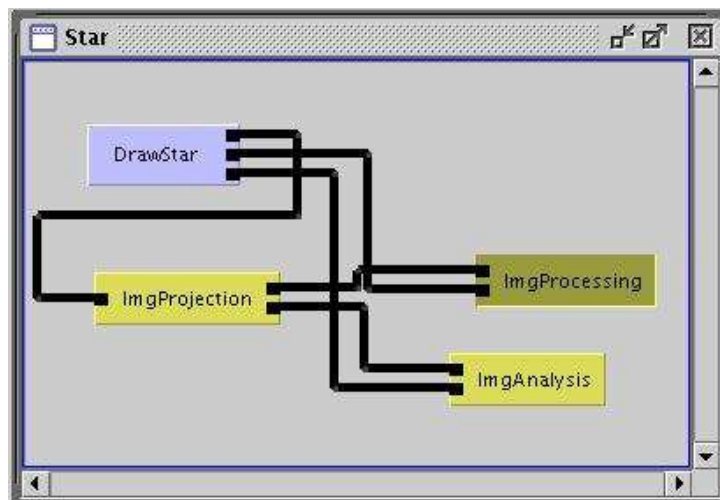


Figure 7.61: The *Star Structural Pattern* supporting the configuration of the *Galaxy* example.

The fully instantiated *Star Structural Pattern*, supporting the configuration of the *Galaxy* example, can be seen in Figure 7.61.

To run the pattern controllers for both the *ImgProcessing* and *ImgAnalysis* pipelines, the main script activates the connections to the Star's pattern controller (*Activate ImgProcessing* in line 18, and *Activate ImgAnalysis* in line 27) and triggers them (*TriggerUnit ImgProcessing* in line 28, and *TriggerUnit ImgAnalysis* in line 29).

The final action in the main script is the application of the *Start* Behavioural Operator over the entire Star pattern (line 30). Due to the active trigger nodes in *EnhContrast* and *CountBlobs*, the user may observe each image individually at the desired pace, as well as the correspondent number of particles in the image.

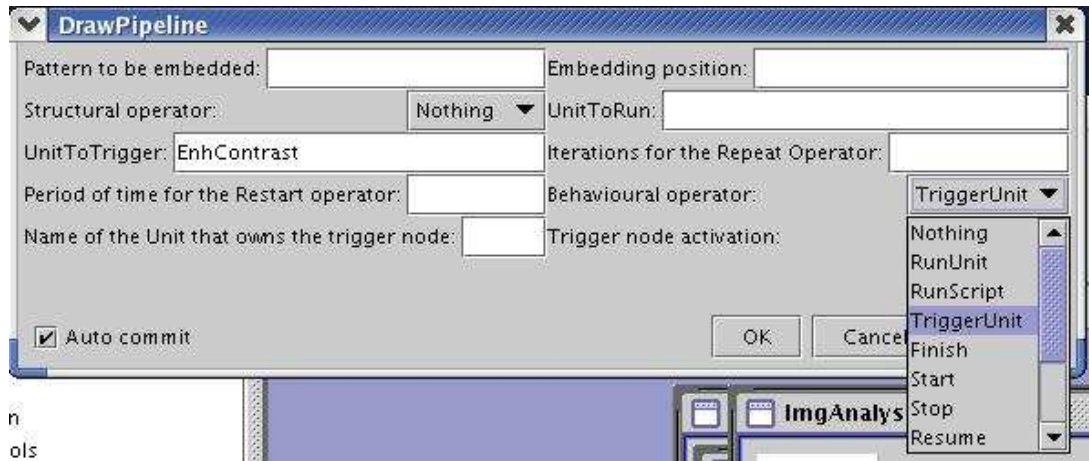


Figure 7.62: Triggering the *EnhContrast* unit to consume the next 2D image.

Figure 7.62 shows how to trigger the *EnhContrast* unit through the operation's panel of the *ImgProcessing* pipeline's pattern controller.

Figure 7.63 shows a full image of the operator panels of the *ImgProcessing* and *ImgAnalysis* pipelines, namely for triggering the named units.

Figures 7.64 and 7.65 show two successive execution steps of the Galaxy example. The image in 7.65 was obtained after triggering both the *EnhContrast* and the *CountBlobs* units, showing the image, and its number of particles, following the image presented in Figure 7.64. The user may, at any time, deactivate the trigger nodes of both units and allow the execution to continue until all frames are processed.

The described independent control of the execution in both pipelines is useful, in this way, to observe a step-by-step execution of tools *ImageView* and *ConstView*. However, their execution is not really independent. Even if the user triggers the *EnhContrast*'s trigger node continuously, the execution of *ImageView* is not allowed to proceed more than once, unless the user triggers the *CountBlobs* unit also continuously. The reason for this dependence is due, mainly, to the Triana's underlying dataflow Behavioural Pattern. First, trigger nodes are "mandatory" as described previously. For example, when an image arrives in the *CountBlobs*'s data node, this unit will only run if its trigger node is triggered by the user. Second, the Triana's channels which connect the units do not buffer more than one independent data item of a data stream. As a consequence, when data is sent to the *CountBlobs*'s data input node, the channel is "full", since the image is not consumed as long as that unit is not triggered. Finally, the *ViewPointProjection* unit uses a Triana's operation to send data to all output nodes, and this forces the blocking of

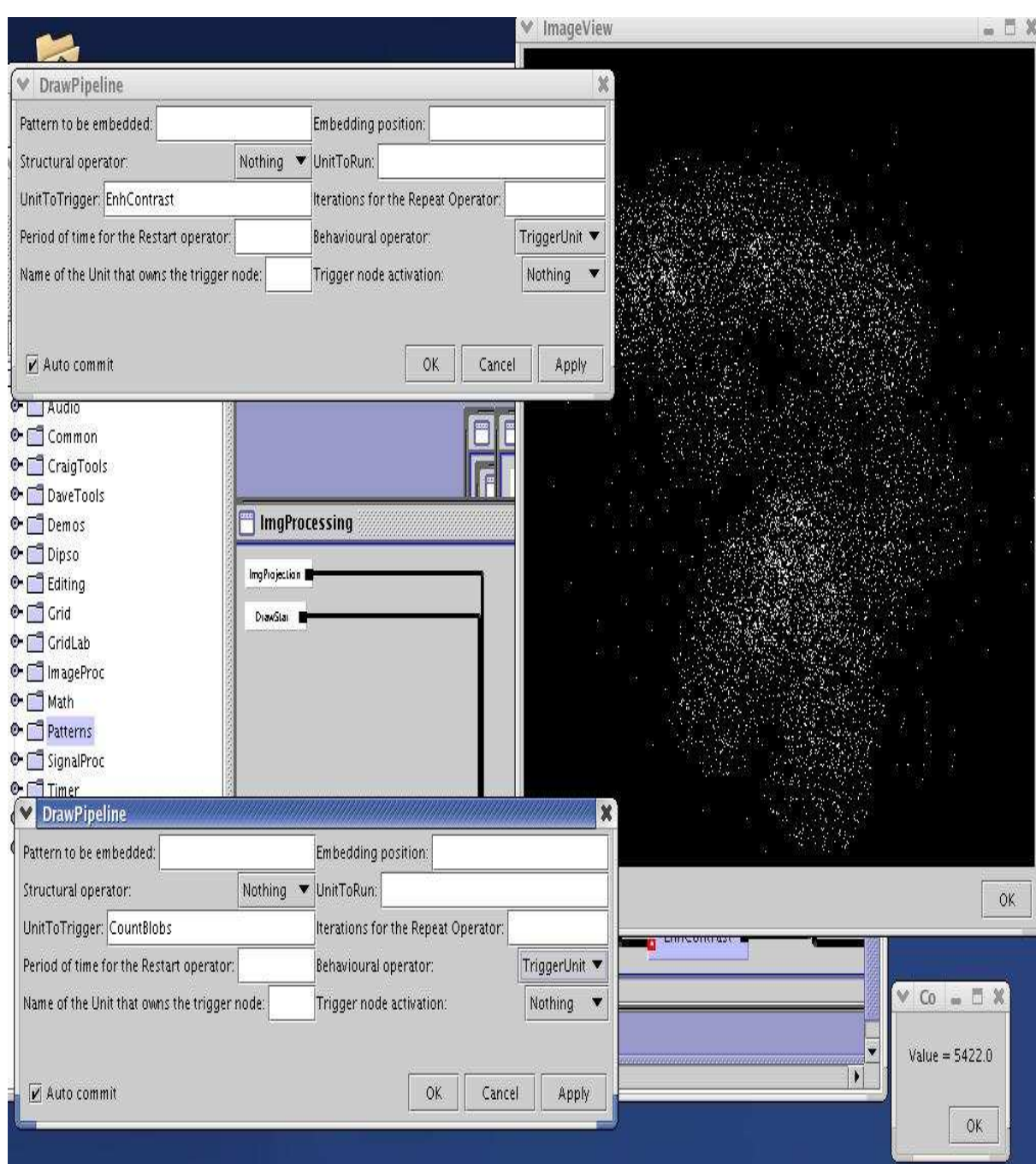


Figure 7.63: Triggering the *EnhContrast* and *CountBlobs* units.

that unit in case data may not be sent through one of those nodes. Consequently, the *ViewPointProjection* unit is “blocked” when outputting the next image since the channel, which connects one of its output data nodes to one of the *CountBlobs* input nodes, is full. As such, the image is not sent to the *EnhContrast*’s unit. The solution would be to define some or all output nodes of *ViewPointProjection* as non-mandatory (named as *optional* in Triana), meaning that the impossibility to send data to one of the nodes would not prevent the image to be sent to the other ones. However, this would imply that some images would be not processed by the downstream units connected to those non-mandatory nodes.

To conclude, and as a result of the not really independent execution of the *ImgProcessing*

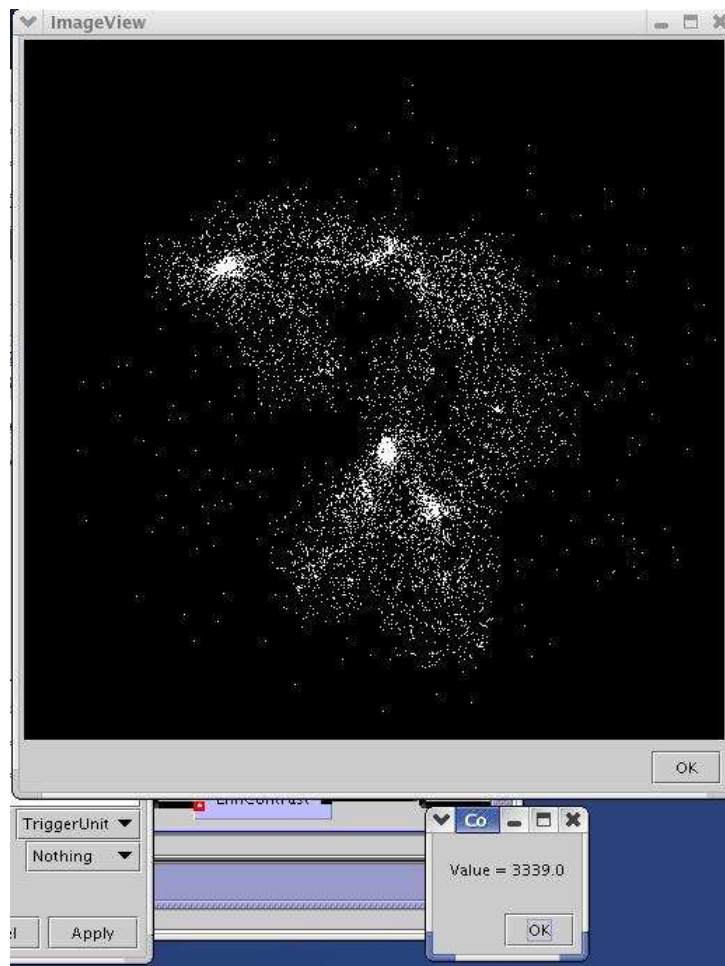


Figure 7.64: An execution step of the Galaxy example.

and *ImgAnalysis* pipelines, a similar step-by-step execution to the one described in Figures 7.64 and 7.65 could be obtained by activating a trigger node upstream. Specifically, the *ViewPoint-Projection* trigger node could be activated (see Figure 7.66).

Behavioural Reconfiguration

The configuration of the Galaxy example with the trigger nodes described in this section 7.6.2 supports a simple behavioural reconfiguration. Specifically, it would be possible to simulate the the *Client/Server* Behavioural Pattern at both the *ImgProcessing* and *ImgAnalysis* pipelines. For example, in the *ImgProcessing* pipeline, the *EnhContrast* could act as a Server and the *ImageView* could become a Client. To implement such a Behavioural Pattern, the end of each iteration at *ImageView* would imply that the *ImgProcessing*'s pattern controller would trigger the *EnhContrast* tool. As such, after completing the display of one image at the client *ImageView*, it would cause another (automatic) request to the server *EnhContrast* to reply with the next image.

Using the same mechanism, a limited version of the *Producer/Consumer* Behavioural Pattern could also be simulated, if a *Buffer* tool would have been added to the pipeline (with the *Increase Structural Operator*) between the *EnhContrast* and *ImageView* tools. The definition of *EnhContrast* as a Producer and *ImageView* as a Consumer by the user, would cause the *ImgProcessing*'s pattern controller to trigger the *EnhContrast* tool a number of times equal to the

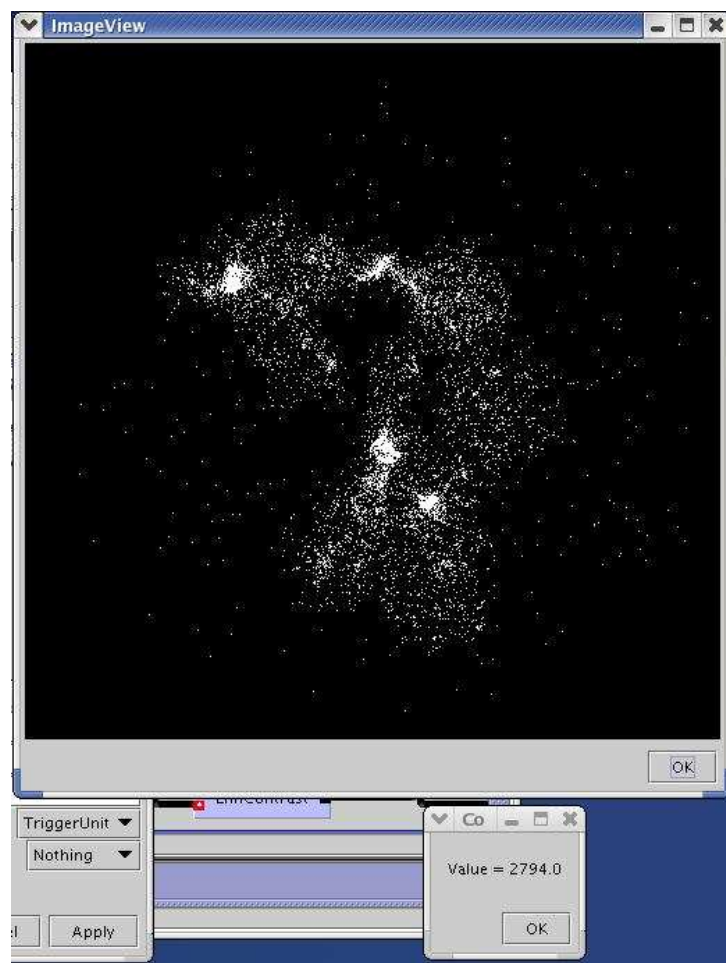


Figure 7.65: The results at both satellites in the next execution step after the one presented in Figure 7.64.

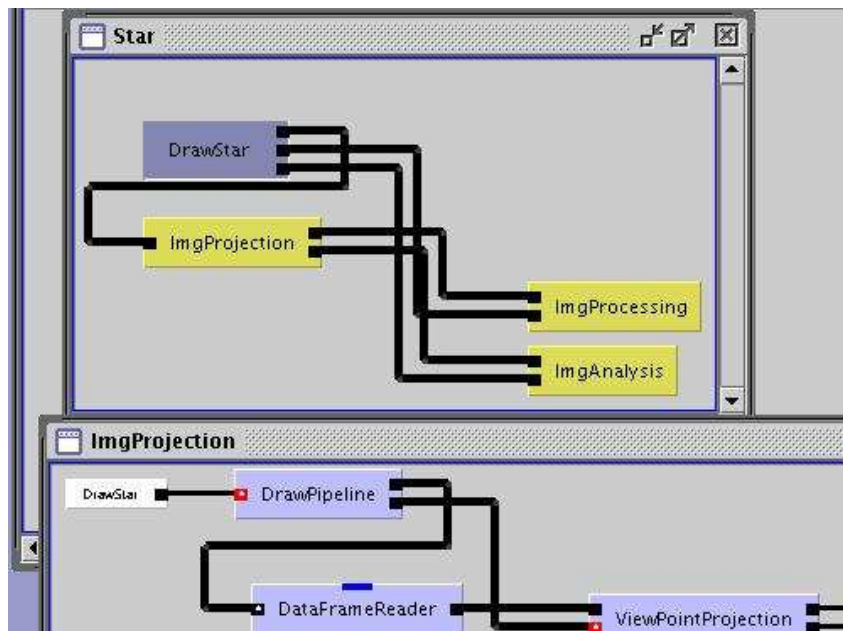


Figure 7.66: The configuration of the *ImgProjection* pipeline for a step-by-step execution.

maximum capacity of the *Buffer* tool. A number of 'n' images would then be produced by *EnhContrast* and saved in *Buffer*. The pattern controller would then run the *Buffer* tool with the *RunUnit* operation, which would send the first image to be consumed by *ImageView*. When completing the display of this image, on the detection of end of execution of *ImageView*, the pattern controller would again run the *Buffer* unit to allow *ImageView* to consume the next image. This sub-process would be repeated 'n'-1 times until the *Buffer* would become empty. Next, the pattern controller would trigger *EnhContrast* 'n' more times, and the main process would be repeated. Finally, the end of execution would be detected when the *ImgProjection*'s pattern controller would inform the *ImgProcessing*' pattern controller of the end of execution of both the *DataFrameReader* and *ViewPointProjection* tools.

Structural Reconfiguration

The configuration of the Galaxy example with the trigger nodes, as described in this section (7.6.2), also supports a useful structural reconfiguration, while the Star's execution is still active. When running the step-by-step execution mode provided by the configuration, the user may decide that it might be useful to add an extra satellite to the star. For example, to save the images produced by the *ViewPointProjection* tool, the user might instantiate the new satellite with the *WriteGif* tool. In case some other image processing would be required, the user might instantiate the satellite with another pipeline. In this way, a structural reconfiguration, independent from a behavioural reconfiguration, is possible while the current execution has not yet finished.

7.7 Simulating Flexible Information Retrieval and Processing

Like in Astrophysics, many other scientific areas need to manipulate large amounts of data. Environmental and Life Sciences, Nuclear Physics, or Earth/Ocean Surface Topography from Space, require the distributed storage of data across different organisations, and their manipulation by many users. Several service-oriented Grids provide customisable or application-specific Portals/Problem Solving Environments in order to facilitate the management and sharing of such data. One common characteristic of the aforementioned environments is database inquiry, where data may be spread over several databases. Many of those environments also provide simulation tools, or even scientific instruments producing data in real time. The present example is a possible simplified configuration of those scientific environments, aiming at clarifying the usefulness of enabling flexible manipulation of Structural Patterns through Operators.

7.7.1 Database Access

The first part of the example outlines a common configuration in the described Grid environments where a client application requires previously stored information to be displayed for analysis. This example makes use of the Facade Structural pattern. However, such is just a

simulation example since the non-topological Structural Patterns were not fully implemented yet.

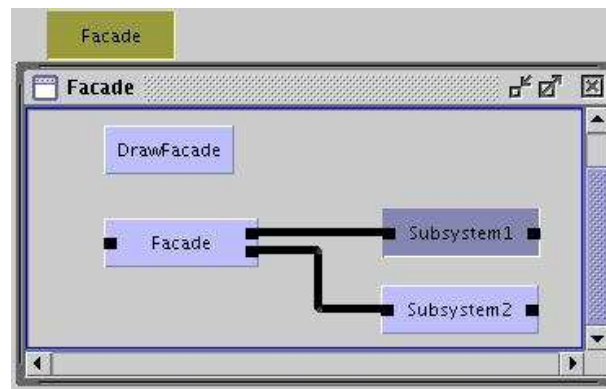


Figure 7.67: *The Facade Structural Pattern Template.*

The Facade, as illustrated before, provides a simple interface to access a set of possibly complex sub-systems (Figure 7.67 shows a Facade Pattern Template). In the present case, namely the access to several databases, either replicated, or providing different types of information, the Facade is a useful design pattern to provide a simplified uniform interface to inquire those databases. Behaviour associated to the Facade may, for example, redirect the requests to the suitable database, either based on the contents of the requests, or on Quality of Service issues like response time in the case of replicated databases.

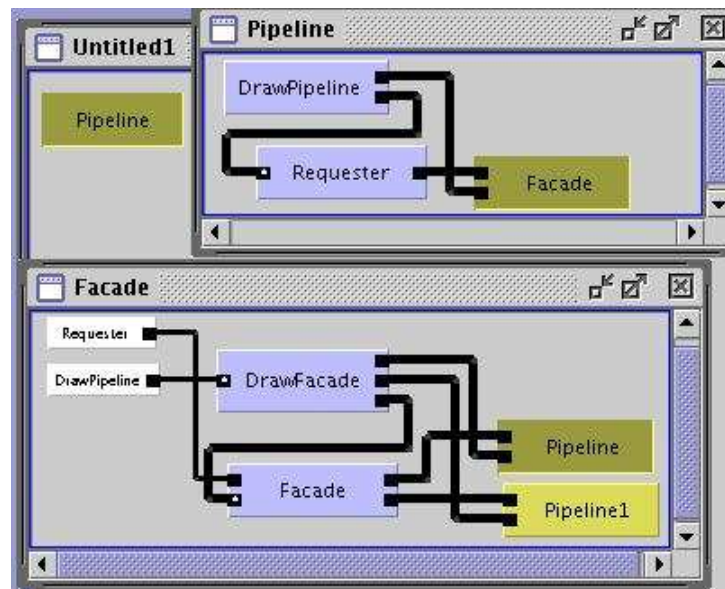


Figure 7.68: *Configuration supporting the request of information to two sub-systems.*

Figure 7.68 shows the Pipeline Structural pattern connecting the client application, the *Requester*, to the Facade Structural pattern. The latter redirects requests to two subsystems already instantiated with two Structural Patterns, namely, *Pipeline* and *Pipeline1*. Both pipelines configure possible associations of databases to data analysis/processing tools. In terms of behaviour, a simple version of the Client/Server Behavioural Pattern may represent the data and control

flows between the *Requester* (client) and the *Facade* (server): the server analyses the requests and redirects them to the subsystems. Additionally, the Producer/Consumer or the Master/Slave are two eligible Behavioural patterns to represent the data and control flows between the Facade and the two subsystems.

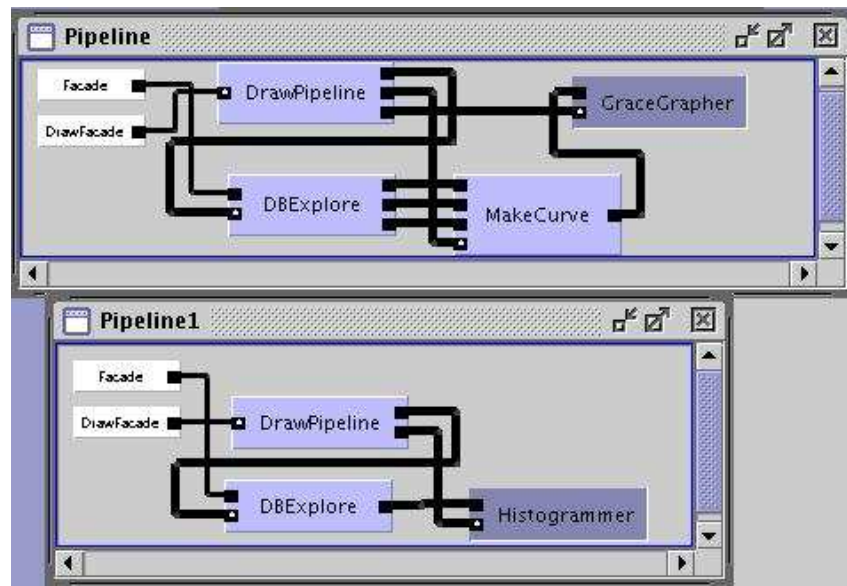


Figure 7.69: Two Pipeline Structural Patterns supporting the configuration of two sub-systems for database access and output data analysis/processing.

The *Pipeline* subsystem in Figure 7.69 contains, as its first element, the *DBExplore* tool to inquire a database through standard SQL [194] requests. As seen in the Figure, three values may be output from the tool, and for these to be visualised in a three dimensional tool, the values are processed by the *MakeCurve* tool and displayed with the *GraceGrapher* tool. The *Pipeline1* subsystem, in turn, provides access to a different or a replicated database combined with another visualisation tool (*Histogrammer*) for output data analysis. This subsystem shows the information in the database according to some criteria. Data and control flows in both pipelines may be provided by the Streaming Behavioural pattern.

The parameter panel of the *DBExplore* tool (Figure 7.70) allows the access to different remote machines, the selection of different databases, and the definition of the query (e.g. *process_id=2*). The Facade hides some of these options, by fixing the databases which are to be accessed, and providing an interface to the *Requester* such that it only has to define the search criteria.

7.7.2 First Structural Reconfiguration: Accessing a New Tool

Based on the configuration shown in Figures 7.68 and 7.69, the user may decide that it would also be necessary to access data available in real time. Requests would either be satisfied by analysing predated or the most recent data, or both. For example, in the case of *Earth Surface Topography from Space*, it is important to evaluate the damages caused by natural disasters like major earthquakes or tsunamis. The comparison of images, before and after the natural disaster, does provide invaluable information concerning the dimension of the damages. Therefore, the user might apply a reconfiguration operation to the previous example, in order to support

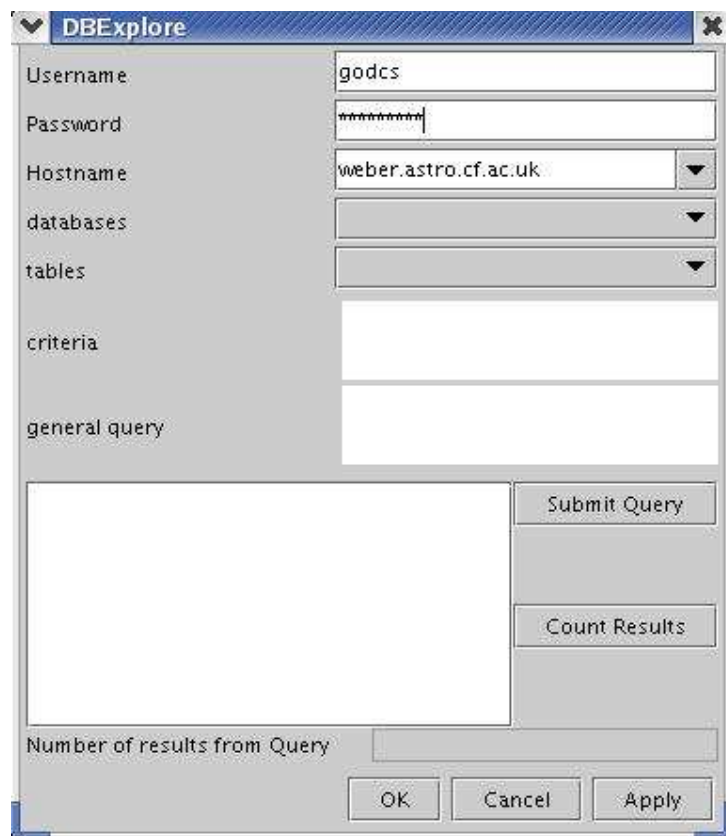


Figure 7.70: The parameter panel of DBExplore, a database inquire tool available in Triana.

the access to a *Real Time Engine* that would gather and process the relevant information (like real time data of the earth surface topology of a specific area).

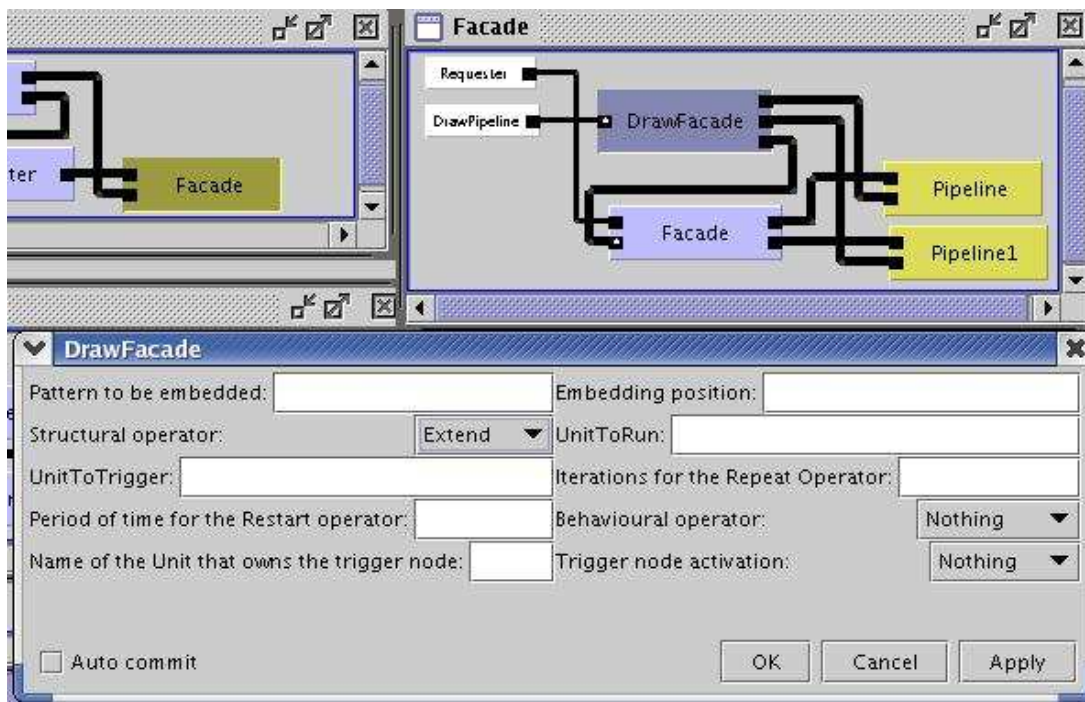


Figure 7.71: Application of the Extend Structural Operator to the Facade Structural Pattern.

In order to accomplish such reconfiguration, the user must first apply the *Stop* Behavioural Pattern to the outmost Pipeline Structural pattern supporting the communication between the *Requester* and the *Facade*. In this way, possible requests from the client are temporarily suspended.

Next, the user applies the Extend Structural Operator to the Facade Structural Pattern (Figure 7.71) to provide a new interface that may redirect requests to the previous databases, to a Real Time Engine, or both. As a result, the pre-existing Facade becomes a subsystem of the new Facade, and the pipeline containing the Real Time Engine becomes the other (new) subsystem. In this way, the previous interface to both databases is not changed, and the outmost facade incorporates this interface and also gives access to the Engine. Behaviour associated with the outmost Facade may hence redirect requests to the innermost Facade, to the Engine, or to both of them. The Client/Server Behavioural Pattern may be used to define the data and control flows between the outmost Facade and its own subsystems.

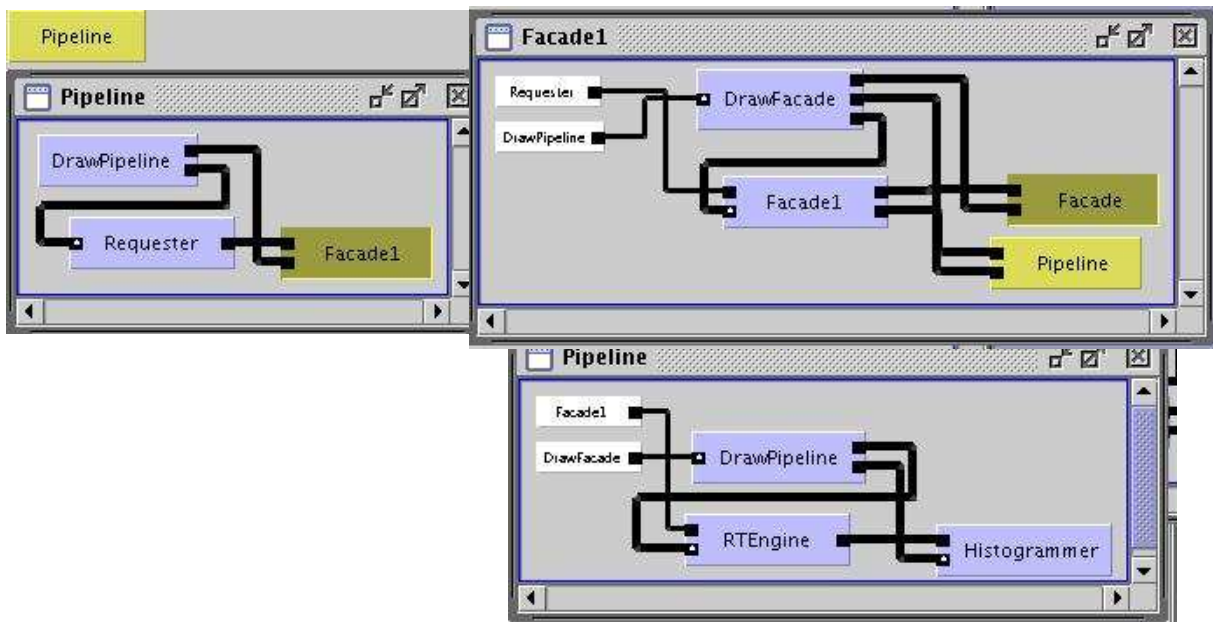


Figure 7.72: Result configuration of the action in Figure 7.71.

Figure 7.72 displays the result configuration consisting of the outmost Facade (*Facade1* in the Figure) which is embedded in the second stage of the outmost pipeline, and the *Requester* remains the first stage. The outmost pipeline is presented on the left-hand side of Figure 7.72. The *Facade1* Structural Pattern is displayed on the top of the right-hand side of the Figure along with its subsystems: *Facade* and *Pipeline*. The *Pipeline* subsystem is displayed on the bottom of the right-hand side of that Figure showing the connection between a Real Time Engine (*RTEngine*) and a visualisation tool (*Histogrammer*).

Figure 7.73, in turn, displays the contents of the innermost Facade, namely *Facade* in Figure 7.72. The subsystems of *Facade* are kept unchanged and can be recalled from Figure 7.69 in the previous subsection.

Finally, to complete the reconfiguration, the user applies the *Resume* Behavioural Pattern to the outmost Pipeline Structural Pattern allowing requests from the *Requester* to the outmost Facade.

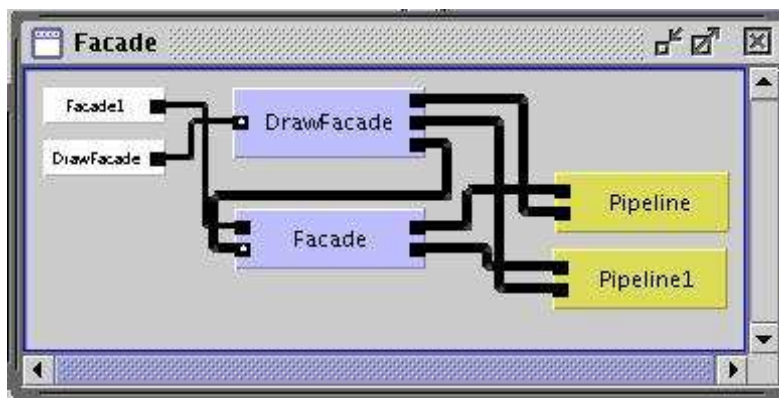


Figure 7.73: The innermost Facade acting as a subsystem of Facade1 in Figure 7.72.

7.7.3 Second Structural Reconfiguration: Pattern Replacement

To further illustrate other possible structural reconfigurations, one might assume another scenario related to the example of a natural disaster as described in the previous subsection. Collected data in the domain of *Earth Surface Topology from Space* is frequently used to build models for catastrophe simulations. Such simulators might be useful, for example, to predict further damages in case a earthquake is followed by some replicas. After the main disaster, the user might want to access such a simulator to predict which areas are more vulnerable. Therefore, the user may apply the *Replace* Structural Operator to the *Facade1* pattern in Figure 7.72 so that another pattern giving access to a simulator becomes the second stage of the outmost (main) pipeline.

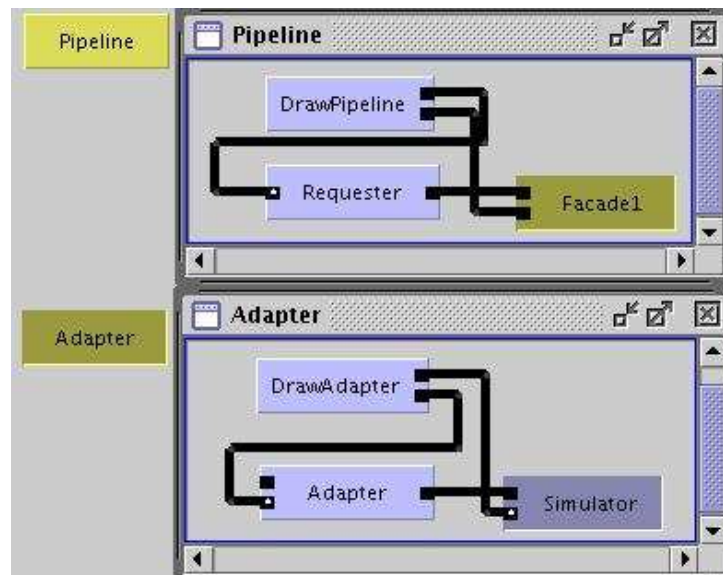


Figure 7.74: An Adapter Structural pattern providing access to a simulation tool. The Adapter pattern will replace Facade1.

In order to make the simulator accessible by the *Requester*, the Adapter Structural pattern transforms the client requests and submits them to the *Simulator* tool. Figure 7.74 shows the Adapter pattern template already instantiated. After the application of the *Replace* operator,

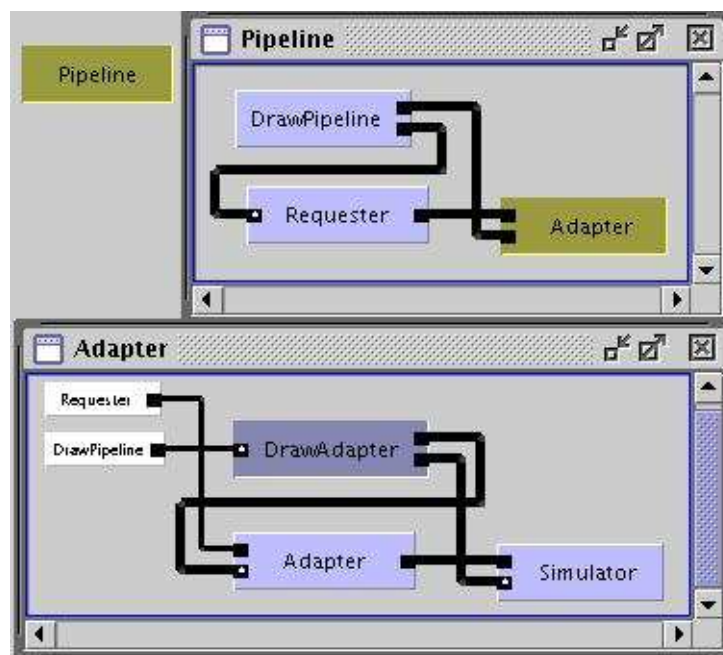


Figure 7.75: The configuration after the application of the Replace Structural Operator described in Figure 7.74.

such Adapter pattern will substitute the Facade1 pattern as the second stage of *Pipeline*. The result of such replacement is shown in Figure 7.75.

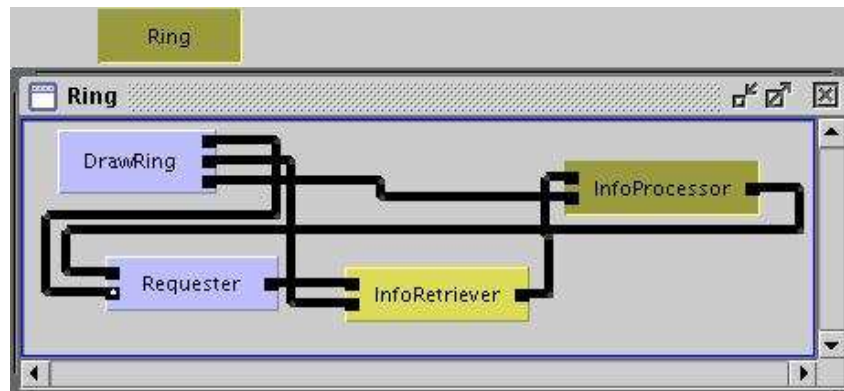


Figure 7.76: Another possible configuration where the client application, the *Requester*, receives processed data it has requested.

Other possible structural reconfigurations are possible. For example, the user might simply want to get information (e.g. about the disaster event) from a Web Service, to process that data to obtain relevant information, and feed the results back to the *Requester* (acting on behalf of the user), therefore closing the cycle. As such, the user may reshape the *Pipeline* pattern in Figure 7.75 into a *Ring* Structural pattern (Figure 7.76)⁴. The *Adapter* pattern in Figure 7.75 may

⁴Although the *Reshape* operator was defined not to support Pattern Instances, the transformation of a *Pipeline* into a *Ring* was supported by a simple implementation of the *Reshape* operator. Namely, the last stage of the *Pipeline* is connected to the first stage forming a *Ring*, independently if the place-holders are already bound or not to Triana's tools/services.

then be replaced with a *InfoRetriever* service (Figure 7.76), and after applying the *Increase Structural* operator to the *Ring*, the new component place holder may be instantiated to a *InfoProcessor* service. Furthermore, the Client/Server Behavioural Pattern would be applied between the *Requester* and the *InfoRetriever*, and the Stream Data-flow would guide the control and data flows of the other *Ring*'s stages. Moreover, the application of the *Restart* Behavioural operator that configuration would allow a periodic information request and processing.

7.8 Summary

This chapter includes some examples of the applicability of the model presented in this thesis, both at: a) the conceptual level including their relevance for distributed and Grid environments; and b) over the Triana tool, which is a Grid-aware workflow-based Problem Solving Environment. The first three examples, namely in sections 7.2, 7.3, and 7.4, are included in the first category. The examples presented in sections 7.5, 7.6, and 7.7 were implemented over the Triana extension with Patterns and Operators as presented in Chapter 6.

The examples illustrate, in our opinion, the adequacy of the model to represent typical applications in distributed and parallel systems in general, and particularly in Grid systems.

8

Conclusions and Future Work

Contents

8.1	Conclusions	290
8.2	Future Work	293

This chapter summarises the main achievements of the research work described in this dissertation, and discuss some still open issues, which will be considered in our future research work.

8.1 Conclusions

The goal of our work was to contribute towards the simplification of the development of Grid applications, namely through an increase of re-usability and flexibility. We aim to achieve this by applying a common model to different stages of the development cycle, namely, to the design, execution, and reconfiguration phases. Our approach is supported by a development methodology providing a set of Design Patterns that can be manipulated through a set of operators.

8.1.1 Contributions of the Thesis

One contribution of this thesis is the proposal of an approach providing Structural and Behavioural Patterns and Operators, where patterns are defined as first-class high-order constructors throughout the whole development cycle. The patterns can be systematically manipulated through operators, either through a visual form or through scripts, and applied by following a proposed methodology. We designate the proposed approach as *a model for pattern- operator-based application development*. Our contribution also comprises a prototype extension of a specific Problem Solving Environment towards the development of a software engineering tool where the composition and orchestrations of Grid resources results from the cited operator-based manipulation of patterns.

Specific Contributions of the Proposed Model

Concerning the characteristics of the model, it is possible to highlight some relevant contributions towards the simplification of application development, namely in Grid environments:

- The model provides support to several stages of an application development cycle, namely from the configuration phase, to the execution and reconfiguration phases. Moreover, the model is uniform throughout those stages. This uniformity results from the persistence of the manipulable reusable abstractions and the way they are act upon during those phases. Specifically:
 - Patterns are the manipulable abstractions and which remain as first class entities during the entire applications' life cycle. As such, patterns are composition entities at the design phase, they are subsequently entities whose execution can be individually controlled, and, finally, patterns can be manipulated individually during the (static/dynamic) reconfiguration phase. The persistence of patterns provides a structured final configuration which is, in this way, amenable to reconfigurations and fine tuned (coordination) control.
 - Operators provide the uniform way to manipulate the variety of patterns as first class entities, and their diversity provide useful actions for each stage. Operators provide consistent refinement during the design and reconfiguration phases, and during execution time the available operators provide execution control without disrupting the overall behavioural semantics of the final configuration.

- The model provides flexibility on application configuration and control by defining two clearly separated dimensions: structure and behaviour.

The existence of Structural and Behavioural Patterns provides different combinations between them. Namely, the same Behavioural Pattern may be applied to different Structural patterns; and the same Structural pattern may give support to different Behavioural Patterns at the same or different times. The selected Structural Patterns include both common topologies as well as design archetypes (e.g. Facade and Proxy Design Patterns). The selected Behavioural Patterns aim to represent common orchestration models in distributed systems and in Grid computing.

The existence of Structural and Behavioural Operators also confers flexibility as structure can be manipulated independently from behaviour and vice-versa.

- Moreover it is possible to reconfigure both the structure and the behaviour during the entire application development cycle, meaning that reconfiguration is possible also at run-time:

Reconfiguration at development time Structural Operators may be used to modify a pattern's configuration either a Pattern Template or a (partially or fully instantiated) Pattern Instance. Additionally, the new added elements may be automatically annotated with a pre-defined behaviour in the particular case of *Regular Patterns*. Namely, a new added element is ruled by the same behaviour as other pre-existent elements in the Patterns. Coordination (Behavioural) Operators, in turn, may modify the behaviour of particular elements within a pattern. Additionally, the behaviour of all elements in a *Regular Pattern* may also be modified through a single operator.

Reconfiguration at run-time The user may reconfigure a running application in two ways:

1. without suspending its execution; such dynamic reconfiguration is limited to *Regular Pattern Instances* (e.g. to add new elements with pre-defined behaviour);
 2. by suspending part of it as a result of the application of an Execution operator (*Stop*) to some selected Pattern Instances. In this case, additional structural and behaviour modifications are also possible.
- The model induces a methodology which aims at simplifying application construction. On one hand, the methodology may guide a less experienced user on programming and controlling an application based on patterns. On the other hand, the methodology is also a systematic approach for both more and less experienced users as the methodology may be systematized into scripts. Specifically, the creation of patterns' instances and operator application may be defined through scripts.
 - The model is suitable both for application development and for service architecture definition. The model's properties such as reusability, extensibility, reconfigurability, and

systematisation, assist both types of developers on manipulating characteristic configurations, either at the application level or on configuring middleware platforms. Nevertheless, such has to be further validated in future work.

Contributions Related to the Model Evaluation

The model was developed in an incremental way and validated by experimentation, instead of being developed through a formal approach only. Although the result represents, in our opinion, a significant set of entities and their interactions, and justifies its relevance, we cannot assume that the model is minimal or complete, but we can claim that it is amenable to extensions.

Towards affirming the suitability of the model, this work presents the following contributions:

Specification of the structure and semantics of the model. The specification included:

- the modelling of common topological schemes using the UML modelling language [68];
- the operational semantics of Structural Operators illustrated through examples;
- a simplified definition of the semantics of some Behavioural Operators using Object-Oriented Petri nets [32];
- the description and illustration of pattern manipulation throughout an application's life cycle by the application of Structural and Behavioural Operators towards reconfiguration.

Implementation support Implementation-wise, this thesis proposed three contributions:

- Identification of a logical layered architecture to support the model.
- Development of a working prototype of the model integrated into a Grid-aware computing environment. Specifically, the model was partially implemented by extending an existing PSE, namely the Triana environment [20], with some relevant patterns and operators. Triana is a Grid-aware and workflow-based Problem Solving Environment which provides an extensible component based interface for composing services in different scientific areas, guaranteeing a sound support for distributed execution. The prototype does not yet provide support for most of the reconfiguration dimensions in the proposed model.
- Analysis of a mapping of some Behavioural Operators onto the *Distributed Resource Management Application API* [43], a distributed resource manager API which supports execution control.

Validation Evaluation of the model for the specification of common application scenarios in Grid environments, and in particular, in the context of Problem Solving Environments.

Particularly, one of the main goals of this work was to contribute to improving existing tools and methodologies supporting Grid application development, based on the Problem Solving Environment approach. In order to achieve this goal, it was considered important to be able to perform experimental assessment and validation of the model and associated methodology, and this required the development of an experimental prototype. As a consequence of this goal, an incremental approach was followed that allows further extensions to be added to the prototype, depending on their relevance to support common recurrent Structural and Behavioural patterns, although we have analysed what we believe are the most common structures and behaviours which are typically found in Grid/distributed applications.

We feel that the above incremental methodology for the experimental validation of the model is, in itself, an important aspect of this proposal, as it enables further evolution and flexible adaptation of the underlying prototype to future situations.

8.2 Future Work

Our initial contribution, presented in this work, has opened the way for further research concerning structured composition and dynamic reconfiguration of Grid applications.

However, several concepts in our approach were not fully validated yet and, therefore, it is our intention to further extend the implementation over the Triana environment, namely,

- the full implementation of the non-topological Structural Patterns;
- the implementation of different Behavioural Patterns;
- the study and implementation of the coordination issues inherent to Hierarchical Pattern Instances ruled by different Behavioural Patterns;
- the implementation of all proposed Structural and Behavioural Operators.
- the validation through application examples that access Web and Grid Services, made accessible in recent Triana versions.

We also acknowledge the addition of new patterns, namely other state-of-the art generic Design Patterns (e.g. the *Decorator* Pattern representing the dynamic addition of new functionalities to an object [9]), or specific patterns, namely for parallel programming [218], workflow systems [79], and Grid systems [47].

Additionally, it is our desire to evaluate the applicability of our approach to different kinds of applications that are more independent of a dataflow model and where coordination control is more complex. To this extent, we also aim to implement our approach over a (Grid-aware) distributed environment that may support more elaborated control mechanisms for control flow independently from data flow (e.g. similarly to some workflow tools giving access to the Globus System like GridAnt [205]/Karajan [206] supported by the Java CoG Kit). Moreover, we envision the importance of manipulating different Coordination/Behavioural Patterns, also organised in hierarchies, through operators. Such patterns may even not be explicitly combined with Structural Patterns. This behaviour-only orchestration in distributed environments may

be useful to less experienced users not interested on how the mappings of those behaviours is supported in practice.

Finally, one important characteristic of our approach, as cited above, is its suitability for service architecture definition, besides application development. Therefore, in our future research we intend to validate the model's properties such as reusability, extensibility, reconfigurability, and systematisation, on manipulating characteristic configurations for building middleware platforms. For example, the possibility of including pattern-based dynamic reconfiguration in those platforms to be automatically triggered by the middleware in the presence of some events. Such may prove to be a contribution to autonomic computing systems research.

Bibliography

- [1] Grid Computing Environments Working Group. See Web site at: <http://www.computingportals.org/>.
- [2] A. Hoheisel, "Fraunhofer Resource Grid – Grid Application Definition Language", Global Grid Forum, Edinburgh, July 2002
- [3] Dan A. Marinescu, "Internet Based Workflow Management: Towards a Semantic Web ", Wiley, New York, 2002
- [4] Dan A. Marinescu, "A Grid Workflow Management Architecture", Global Grid Forum Working Document (submitted). School of Electrical and Computer Engineering, University of Central Florida, Orlando, Florida 32816, USA
- [5] J. McLaren, V. Sander, W. Ziegler, "Grid Resource Allocation Agreement Protocol (GRAAP)". See web site at: <http://www.people.man.ac.uk/~zcgujm/GGF/graap-wg.html>.
- [6] Craig Lee and Domenico Talia, "Grid Programming models: current tools, issues and directions", In Grid Computing: Making the global infrastructure a reality. Fran Berman, Geoffrey Fox and Tony Hey (eds), Wiley, 2003.
- [7] O. F. Rana and D. Jennings, "Automating Performance Analysis from UML Design Patterns" (Research Note), Proceedings of EuroPar 2000, Munich, Germany
- [8] UML Tools. See Web site at: http://www.cetus-links.org/oo_uml.html#oo_uml_utilities_tools.
- [9] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1994.
- [10] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, "Pattern-Oriented Software Architecture: A System of Patterns", John Wiley & Sons, 1998.
- [11] I. Foster, C. Kesselman, "The Globus Project: A Status Report", Proc. IPPS/SPDP '98 Heterogeneous Computing Workshop, pp. 4-18, 1998. Globus related publications can also be obtained from Web site at: <http://www.globus.org/research/papers.html>.
- [12] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, "Grid Service Specification", Open Grid Service Infrastructure WG, Global Grid Forum, Toronto, Canada, February 2002.

- [13] G. von Laszewski, I. Foster, J. Gawor, and P. Lane, "A Java Commodity Grid Kit", *Concurrency and Computation: Practice and Experience*, pages 643-662, Volume 13, Issue 8-9, 2001.
- [14] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, S. Tuecke, "A Resource Management Architecture for Metacomputing Systems", *Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, pg. 62-82, 1998. See web-site at <http://www.globus.org>.
- [15] Rajkumar Buyya, "Grid Computing InfoCentre". See Web site at: <http://www.gridcomputing.com/>.
- [16] G. Fox, D. Gannon, and M. Thomas, "A Summary of Grid Computing Environments", *Concurrency and Computation: Practice and Experience*, 2002.
- [17] Open Grid Forum (OGF), formerly Global Grid Forum (GGF). See Web site at: <http://www.gridforum.org/>.
- [18] D.W.Walker, M. Li, O.F.Rana, M.Shields, Y.Huang, "The Software Architecture of a Problem Solving Environment", *Concurrency: Practice and Experience*, December 2000.
- [19] D. Abramson et al., "A Tool for Distributed Parametric Modelling". See Web site at: <http://www.csse.monash.edu.au/~davida/nimrod.html/>.
- [20] I. Taylor et al., "TRIANA". See Web site at: <http://www.triana.co.uk/>.
- [21] I. Taylor, O. F. Rana, R. Philp, I. Wang, M. Shields, "Supporting Peer-2-Peer Interactions in the Consumer Grid", 8th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS) at IPDPS, Nice, France, April 2003. IEEE Computer Society Press.
- [22] C.Lee, S.Matsuoka, D.Talia, A.Sussman, N.Karonis, G.Allen, and J.Saltz, "A Grid Programming Primer", *Programming Models Working Group, Global Grid Forum meeting*, Washington DC, July 16-18, 2001.
- [23] Jia Yu, Rajkumar Buyya, "A taxonomy of scientific workflow systems for grid computing", *SIGMOD Record* 34(3): 44-49, 2005.
- [24] S. Gorlatch, "Extracting and implementing list homomorphisms in parallel program development", *Science of Computer Programming*, 33(1), pp 1-27, 1998.
- [25] H. Bischof, S. Gorlatch, E. Kitzelmann. Cost Optimality and Predictability of Parallel Programming with Skeletons. *Euro-Par 2003, LNCS 2790*, pp. 682-693 (distinguished paper).
- [26] C. A. Herrmann and C. Lengauer, "Transforming Rapid Prototypes to Efficient Parallel Programs", book chapter in "Patterns and Skeletons for Parallel and Distributed Computing", (Fethi A. Rabhi and Sergei Gorlatch (Eds)), Springer Verlag, 2002
- [27] The Common Component Architecture Forum. See Web site at: <http://www.cca-forum.org/>.

- [28] Franz Achermann and Oscar Nierstrasz, "Applications = Components + Script – A Tour of Piccola", *Software Architectures and Component Technology*, Mehmet Aksit (Ed.), pp. 261-292, Kluwer, 2001.
- [29] E. Friedman-Hill, "The Rule Engine for the Java Platform". See Web site at: <http://herzberg.ca.sandia.gov/jess/>.
- [30] Michael Weber, Ekkart Kindler, "The Petri Net Markup Language (PNML)". See Web site at: <http://www.informatik.hu-berlin.de/top/pnml/>.
- [31] Z. Nemeth and V. Sunderam, "A Formal Framework for Defining Grid Systems", *Proceedings of IEEE CCGrid 2002*, Berlin, Germany.
- [32] D. Buchs and N. Guelfi, "A Formal Specification Framework for Object-Oriented Distributed Systems", *IEEE Transactions on Software Engineering*, Vol. 26, No. 7, July 2000.
- [33] G. Di Marzo Serugendo, D. Mandrioli, D. Buchs and N. Guelfi, "Adding Real-Time Constraints to Synchronised Petri Nets", *Technical report 2000/341*, EPFL, Lausanne, Switzerland, 2000.
- [34] O. F. Rana, D. W. Walker, "Service Design Patterns for Computational Grids", in "Patterns and Skeletons for Parallel and Distributed Computing", F. Rabhi and S. Gorlatch(Eds), Springer, 2002.
- [35] Y. Aridor, and D. B. Lange, "Agent design patterns: elements of agent application design", *Proceedings of the Second international Conference on Autonomous Agents* (Minneapolis, Minnesota, United States, May 10 - 13, 1998), K. P. Sycara and M. Wooldridge, Eds. AGENTS '98. ACM Press, New York, NY, 108-115, 1998.
- [36] J. C. Cunha, P. Medeiros, V. Duarte, J. Lourenco, M. C. Gomes, "An Experience in Building a Parallel and Distributed Problem-Solving Environment", *Proceedings of the Intl. Conference on Parallel and Distributed Processing Techniques and Applications*, PDPTA'99, Las Vegas, USA, CSREA Press, July, 1999.
- [37] C. Gomes, O. F. Rana, J. Cunha, "Pattern/Operator Based Problem Solving Environments", in *Proceedings of the 10th Euro-Par Conference*, Pisa, Italy, August/September 2004, M. Danelutto, D. Laforenza, M. Vanneschi (Eds), Springer.
- [38] J. Cunha, O. Rana, P. Medeiros, "Future Trends in Distributed Applications and Problem-Solving Environments", *Invited Paper. Future Generation Computer Systems*, Elsevier Science (22 pages), 2003 .
- [39] T. Goodale, "Applications and the Grid", *GridLab document*. See Web site at: <http://www.gridlab.org/>.
- [40] I. Taylor, M. Shields, I. Wang, and O. Rana, "Triana Applications within Grid Computing and Peer to Peer Environments", *Journal of Grid Computing*, 1(2):199-217, Kluwer Academic Press, 2003.

- [41] Gabrielle Allen, Kelly Davis, Konstantinos N. Dolkas, Nikolaos D. Doulamis, Tom Goodale, Thilo Kielmann¹, Andr_ Merzky, Jarek Nabrzyski, Juliusz Pukacki, Thomas Radke, Michael Russell, Ed Seidel, John Shalf and Ian Taylor, "Enabling Applications on the Grid: A GridLab Overview", International Journal of High Performance Computing Applications, Special issue on Grid Computing: Infrastructure and Applications, Vol. 17, No. 4, 449-466 (2003).
- [42] The GridLab Project. See Web site at: <http://www.gridlab.org/>.
- [43] Habri Rajic, Roger Brobst et al., "Distributed Resource Management Application API Specification 1.0". Global Grid Forum DRMAA Working Group. See Web site at: <http://www.drmaa.org/>.
- [44] M.C.Gomes, O.F.Rana, J.C.Cunha "Pattern Operators for Grid Environments", Scientific Programming Journal, Volume 11, Number 3, 2003, IOS Press, Editors: R. Perrot and B. Szymanski.
- [45] M.C.Gomes, J.C.Cunha, O.F.Rana, "A Pattern-based Software Engineering Tool for Grid Environments", Concurrent Information Processing and Computing proceedings, NATO Advanced Research Workshop, Sinaia, Romenia, June 2003, IOS Press.
- [46] O.F.Rana, M.C.Gomes, J.C.Cunha, "Patterns and Operators for Grid Software Development", WWW/Internet 2003 proceedings, IADIS International Conference, Algarve, Portugal, 5-8 November, 2003.
- [47] "Patterns and Skeletons for Parallel and Distributed Computing", F. Rabhi and S. Gorlatch (Eds), Springer, 2002.
- [48] The Globus Project, "Open Grid Services Architecture". See Web site at: <http://www.globus.org/ogsa>.
- [49] Ian Foster, Carl Kesselman (Editors), "The Grid: Blueprint for a New Computing Infrastructure", Morgan Kaufmann, 1998.
- [50] Ian Foster, Carl Kesselman (Editors), "The Grid 2: Blueprint for a New Computing Infrastructure", Morgan Kaufmann, 2004.
- [51] The Globus Project. See Web site at: <http://www.globus.org/>.
- [52] The Legion Project. See Web site at: <http://legion.virginia.edu/>.
- [53] The UNICORE forum. See Web site at: <http://www.unicore.org/>.
- [54] B. Wydaeghe, W. Vanderperren, "Visual Composition Using Composition Patterns", Proc. Tools 2001, Santa Barbara, USA, July 2001.
- [55] ObjectVenture, The ObjectAssembler Visual Development Environment. See Web site at: <http://www.objectventure.com/objectassembler.html>.
- [56] The MyGrid Project. See Web site at: <http://www.mygrid.org.uk>.

- [57] The DataGrid Project. See Web site at:
<http://eu-datagrid.web.cern.ch/eu-datagrid>.
- [58] "XCAT 2.0: A Component-Based Programming Model for Grid Web Services", M. Govindaraju, S. Krishnan, K. Chiu, A. Slominski, D. Gannon, R. Bramley, Technical Report Number 562, Indiana University, Bloomington, Indiana, June 2002.
- [59] T. Sandholm and J. Gawor, "Grid Services Development Framework Design", Draft Version 0.13. Available at: <http://esc.dl.ac.uk/WebServices/OGSA/ogsadf.pdf>
- [60] Foster, I., Kesselman, C., Nick, J. and Tuecke, S. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. The Globus Project, 2002. Available at www.globus.org/research/papers/ogsa.pdf.
- [61] J. Bosch, "Design Patterns as Language Constructs", Journal of Object-Oriented Programming, Vol. 11(2), pages 18-32, 1998.
- [62] P. Forbrig and R. Lämmel, "Programming with Patterns", Proc. TOOLS 2000, Santa Barbara, USA, July 2000.
- [63] M. Baker and G. Smith, "Jini Meets the Grid", International Conference on Parallel Processing Workshops, Valencia, Spain, September 2001. IEEE Computer Society.
- [64] N. Furmento, A. Mayor, S. McGough, S. Newhouse, T. Field, and J. Darlington, "An Integrated Grid Environment for Component Applications", In 2nd International Workshop on Grid Computing 2001, volume 2242 of Lecture Notes in Computer Science, pages 26-37, Denver, November 2001.
- [65] A. Denis, C. Perez, T. Priol, and A. Ribes, "Padico: A Component-Based Software Infrastructure for Grid Computing", In 17th International Parallel and Distributed Processing Symposium (IPDPS2003), Nice, France, April 2003. IEEE Computer Society.
- [66] M. Lorch and D. Kafura, "Symphony A Java-based Composition and Manipulation Framework for Computational Grids", Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid, 21 - 24. May 2002, Berlin, Germany.
- [67] D. Webb and A. Wendelborn, "The PAGIS Grid Application Environment", Submitted to 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid'03), Tokyo, Japan, 12-15 May, 2003. Available at <http://www.cs.adelaide.edu.au/darren/files/ccgrid03.pdf>
- [68] G. Booch, I. Jacobson, and J. Rumbaugh, "The Unified Modeling Language User Guide", Addison-Wesley Professional, 1999.
- [69] M. Fowler, K. Scott, "UML Distilled. Applying The Standard Object Modeling Language", Addison-Wesley, 1997.
- [70] Gentleware, The Poseidon UML Tool, Web site at: <http://www.gentleware.com/>.
- [71] ANL, "The Globus System". See Web site at: <http://www.globus.org/>.

- [72] I. Taylor, M. Shields, I. Wang, R. Philp, S. Majithia, "Grid-Aware Triana Prototype", GridLab - A Grid Application Toolkit and Testbed, Work Package 3: Work-Flow Application Toolkit (TGAT). See Web site at: <http://www.gridlab.org/>.
- [73] I. Taylor S. Majithia, M. Shields, I. Wang, "Triana WorkFlow Specification ", GridLab - A Grid Application Toolkit and Testbed, Work Package 3: Work-Flow Application Toolkit (TGAT). See Web site at: <http://www.gridlab.org/>.
- [74] Department of Physics and Astronomy, Cardiff University, Wales. See Web site at: <http://www.astro.cf.ac.uk/>.
- [75] The GEO600 project. See Web site at: <http://www.geo600.uni-hannover.de/>.
- [76] Extensible Markup Language (XML). World Wide Web Consortium (W3C. See Web site at: <http://www.w3.org/XML>.
- [77] Project JXTA. See Web site at: <http://www.w3.org/XML>.
- [78] W.M.P. van der Aalst, A.H.M. ter Hofsted, "Workflow Patterns: On the Expressive Power of (Petri-net-based) Workflow Languages", In K. Jensen, editor, Proceedings of the Fourth Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2002), volume 560 of DAIMI, pages 1-20, Aarhus, Denmark, August 2002. University of Aarhus.
- [79] "Workflow patterns". See Web site at: <http://www.workflowpatterns.com>.
- [80] J. Brzezinski, J. Nabrzyski, J. Puckacki, T. Piontek, K. Kurowski, L. Ludwiczak, R. M. Hapke, Doulamis, A. M. Varvarigos, Strugalski, N. Doulamis, and K. Dolkas, "Technical Specification of the GridLab Resource Management System", <http://www.gridlab.org/Resources/Deliverables/D9.2.pdf>, July 2002.
- [81] GridLab Workpackage 9. See web site at <http://www.gridlab.org/WorkPackages/wp-9>.
- [82] Shalil Majithia, Matthew Shields, Ian Taylor, Ian Wang, "Triana: A Graphical Web Service Composition and Execution Toolkit", IEEE International Conference on Web Services (ICWS'2004), July 2004, San Diego, California, USA.
- [83] Shalil Majithia, Ian Taylor, Matthew Shields, Ian Wang, "Triana as a Graphical Web Services Composition Toolkit", UK e-Science All Hands Meeting 2003, September 2003, Nottingham, UK.
- [84] David Churches, Gabor Gombas, Andrew Harrison, Jason Maassen, Craig Robinson, Matthew Shields, Ian Taylor, Ian Wang , "Programming Scientific and Distributed Workflow with Triana Services", Global Grid Forum 2004 Special Issue of Concurrency and Computation: Practice and Experience. Available at <http://www.cc-pe.net/iuhome/workflow2004index.html>
- [85] UDDI.org UDDI Technical White Paper UDDI.org, September 6, 2000. See website at <http://www.uddi.org>.
- [86] Web Services Invocation Framework (WSIF). See website at <http://ws.apache.org/wsif>.

- [87] F. Curbera, W. Nagy, and S. Weerawarana, "Web Services: Why and How", OOPSLA 2001 Workshop on Object-Oriented Web Services. ACM, 2001.
- [88] R. Khalaf, N. Mukhi, and S. Weerawarana, "Service-oriented Composition in BPEL4WS", WWW - World Wide Web Conference Series, 2003.
- [89] F. Curbera, N. Mukhi, and S. Weerawarana, "On the Emergence of a Web Services Component Model", Proc. of the 6th Workshop on Component-Oriented Programming (WCOP), 2001.
- [90] Elias N. Houstis, John R. Rice, Efstratios Gallopoulos, Randall Bramley (Eds), "Enabling Technologies for Computational Science: Frameworks, Middleware and Environments", Kluwer Academic Publishers, 2000.
- [91] Elias N. Houstis, John R. Rice, "Future Problem Solving Environments for Computational Science", In Mathematics and Computers in Simulation journal, 54, 243-257, 2000.
- [92] E. N. Houstis, J. R. Rice. On the Future of Problem Solving Environments. CSD TR-00-009, Computer Science Department, Purdue University, 78 pp., March 2000.
- [93] E. Gallopoulos, E. Houstis, and J. Rice, "Computer As Thinker/Doer: Problem-Solving Environments for Computational Science", IEEE Computational Science and Engineering, vol. 1, no. 2, pp. 11-23, 1994.
- [94] R. Buyya, T. Eidson, D. Gannon, E. Laure, S. Matsuoka, T. Priol, J. Saltz, E. Seidel, Y. Tanaka, "Problem Solving Environment Comparison White Paper", technical report, February, 2001. Online at http://www.eece.unm.edu/apm/WhitePapers/APM_Sys_Comp.pdf
- [95] C. Lee, D. Talia, "Grid Programming Models: Current Tools, Issues and Directions", In Grid Computing: Making the Global Infrastructure a Reality, Wiley, 2003.
- [96] Frank Berman, Geoffrey C. Fox, Anthony J. G. Hey (Eds), "Grid Computing: Making the Global Infrastructure a Reality", Wiley, 2003,
- [97] Jean Bacon, "Concurrent Systems – Operating Systems, Database and Distributed Systems: An Integrated Approach", Second Edition, Addison-Wesley, 1996.
- [98] Len Bass, Paul Clements, Rick Kazman, "Software Architecture in Practice", Addison-Wesley, 1998.
- [99] David Garlan, Mary Shaw, "An Introduction to Software Architecture", technical report CMU-CS-94-166, Carnegie Mellon University, January 1994.
- [100] Clemens Szypersky, "Component Software: Beyond Object-Oriented Programming", Addison-Wesley, 1998.
- [101] Sanjiva Weerawarana, Francisco Curbera, Matthew J. Duftler, David A. Epstein, Joseph Kesselman, "Bean markup language: a composition language for JavaBeans components", Proceedings of the 6th conference on USENIX Conference on Object-Oriented Technologies and Systems - Volume 6, 2001.

- [102] George Coulouris, Jean Dollimore, Tim Kindberg, "Distributed Systems: Concepts and Design", Addison-Wesley, 2001.
- [103] Philip Eskelin, "Component Interaction Patterns", Proceedings of 6th Conference on the Pattern Languages of Programming (PloP), Illinois, USA, 1999.
- [104] JavaBeans technology, See website at <http://java.sun.com/products/javabeans/>.
- [105] CORBA, Object Management Group (OMG), See website at <http://www.corba.org/>.
- [106] .NET framework, Microsoft, <http://msdn2.microsoft.com/en-us/netframework>.
- [107] Java Remote Method Invocation, Sun Microsystems, <http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp>.
- [108] Interface Definition Language, OMG group, See website at http://www.omg.org/gettingstarted/omg_idl.htm.
- [109] Nicholas Carriero, David Gelernter, "Linda in Context", Communications of the ACM, Volume 32, Issue 4, Pages: 444 - 458, 1989.
- [110] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann, "Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects", John Wiley and Sons, 2000.
- [111] The Enhance Project, See website at <http://groups.inf.ed.ac.uk/enhance/>.
- [112] M. Hiltunen and R. Schlichting, "The Cactus approach to building configurable middleware services", Proceedings of the Workshop on Dependable System Middleware and Group Communication (DSMGC), 2000.
- [113] "SCIRun: A Scientific Computing Problem Solving Environment, Scientific Computing and Imaging Institute (SCI)", <http://software.sci.utah.edu/scirun.html>.
- [114] The XCAT project, Indiana University, <http://www.extreme.indiana.edu/xcat/index.html>.
- [115] CO2P3S (Correct Object-Oriented Pattern-based Parallel Programming System), See website at <http://www.cs.ualberta.ca/systems/cops/>.
- [116] John Anvik, Steve MacDonald, Duane Szafron, Jonathan Schaeffer, Steven Bromling and Kai Tan, Generating Parallel Programs from the Wavefront Design Pattern, 7th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'2002) at IPDPS, April 2002, Ft. Lauderdale U.S.A, CD-ROM pp. 1-8. (Best Paper Award).
- [117] S. MacDonald. From Patterns to Frameworks to Parallel Programs. PhD thesis, Department of Computing Science, University of Alberta, 2001.
- [118] Steve MacDonald, John Anvik, Steve Bromling, Jonathan Schaeffer, Duane Szafron, Kai Tan. "From Patterns to Frameworks to Parallel Programs". Parallel Computing, 28(12);1663-1683, 2002.

- [119] K. Mani Chandy and S. Taylor. An Introduction to Parallel Programming. Jones and Bartlett Publishers, 1992.
- [120] R. Johnson. Frameworks = (components + patterns). CACM, 40(10):39-42, 1997.
- [121] K. Tan, D. Szafron, J. Schaeffer, J. Anvik, and S. MacDonald, "Using Generative Design Patterns to Generate Parallel Code for a Distributed Memory Environment", in Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2003), San Diego, CA, June 2003, pages 203-215.
- [122] J. Schaeffer, D. Szafron, G. Lobe, I. Parsons, "The Enterprise model for developing distributed applications Parallel and Distributed Technology", Systems and Applications, IEEE, Volume 1, Issue 3, Page(s):85 - 96, Aug. 1993.
- [123] D. Szafron, J. Schaeffer, P. Iglinski, . Parsons, R. Kornelsen and C. Morrow, Enterprise: Current Status and Future Directions, CASCON 94, CDROM Proceedings, Toronto, October, 1994.
- [124] Steve Bromling, Steve MacDonald, John Anvik, Jonathan Schaefer, Duane Szafron, Kai Tan, Pattern-based Parallel Programming, Proceedings of the International Conference on Parallel Programming (ICPP'2002), August 2002, Vancouver Canada, 257-265.
- [125] Steve MacDonald, Duane Szafron, Jonathan Schaeffer, John Anvik, Steve Bromling, Kai Tan, Generative Design Patterns, 17th IEEE International Conference on Automated Software Engineering (ASE) September 2002, Edinburgh, UK, 23-34.
- [126] J. Bosch. Design patterns as language constructs. JOOP, 11(2), pp. 18-32, 1998.
- [127] F. Budinsky, M. Finnie, J. Vlissides, and P. Yu. Automatic code generation from design patterns. IBM Systems Journal, 35(2), pp. 151-171, 1996.
- [128] G. Florijn, M. Meijers, and P. van Winsen. Tool support for object-oriented patterns. In ECOOP, Vol. 1241 of LNCS, pp. 472-495. Springer, 1997.
- [129] TogetherSoft Corporation. TogetherSoft ControlCenter tutorials: Using design patterns.
- [130] J. Anvik, J. Schaeffer, D. Szafron, and K. Tan, Why Not Use a Pattern-based Parallel Programming System?, EuroPar International Conference on Parallel and Distributed Computing (EuroPar), August 2003, Klagenfurt Austria, pp. 81-86.
- [131] G. Wilson. "Assessing the usability of parallel programming systems: The Cowichan problems". In IFIP Working Conference on Programming Environments for Massively Parallel Distributed Systems, pages 183-193, 1994.
- [132] M. MacNaughton, M. Cutumisu, D. Szafron, J. Schaeffer, J. Redford and D. Parker, ScriptEase: Generative Design Patterns for Computer Role-Playing Games, 19th IEEE International Conference on Automated Software Engineering (ASE) September 2004, Linz, Austria, pp. 88-99.

- [133] Using Generative Design Patterns to Develop Network Server Applications, Zhuang Guo; Schaeffer, J.; Szafron, D.; Earl, P.; Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International 04-08 April 2005.
- [134] Jini Architectural Overview, 2001. <http://www.sun.com/software/jini/whitepapers/architecture.pdf>.
- [135] Sun Microsystems. Java Remote Method Invocation Specification, JDK 1.1, 1997. Available at http://java.sun.com/products/jdk/rmi_ed.
- [136] D.C. Schmidt. Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching. In *Pattern Languages of Program Design*, Addison-Wesley, 1995, pp. 529-545.
- [137] T. Harrison, I. Pyrarli, D. Schmidt, and T. Jordan. Proactor An Object Behavioral Pattern for Dispatching Asynchronous Event Handlers. Washington University Technical Report: WUCS-97-34, 1997.
- [138] T. Harrison and D. Schmidt. Asynchronous Completion Tokens: An Object Behavioral Pattern for Efficient Asynchronous Event Handling. In *Pattern Languages of Program Design*, Addison-Wesley, 1997.
- [139] D.C. Schmidt. Acceptor-Connector: An Object Creational Pattern for Connecting and Initializing Communication Services. In *Pattern Languages of Program Design*, Addison-Wesley, 1997.
- [140] Apache Avalon Project. <http://avalon.apache.org>.
- [141] J. Hu and D. Schmidt. JAWS: A Framework for High Performance Web Servers. In *Domain-Specific Application Frameworks: Frameworks Experience by Industry*, Wiley & Sons, 1999, pp. 339-376.
- [142] M. Fayad, D. Schmidt, and R. Johnson, editors. *Building Application Frameworks*, John Wiley & Sons, 1999.
- [143] S. Siu, M. De Simone, D. Goswami, and A. Singh. "Design patterns for parallel programming". In *Proceedings of the 1996 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 230-240, 1996.
- [144] D. Goswami, A. Singh, and B. Priess. "Using object-oriented techniques for realizing parallel architectural skeletons". In *Proceedings of the Third International Scientific Computing in Object-Oriented Parallel Environments Conference*, volume 1732 of *Lecture Notes in Computer Science*, pages 130-141. Springer-Verlag, 1999.
- [145] Using Object-Oriented Techniques for Realizing Parallel Architectural Skeletons Dhrubajyoti Goswami, Ajit Singh, and Bruno Richard Preiss. In *Proc. ISCOPE '99*, San Francisco, CA, December 1999.
- [146] Dhrubajyoti Goswami, Ajit Singh, and Bruno Richard Preiss. *Building Parallel Applications using Design Patterns*. In *Advances in Software Engineering: Topics in Comprehension, Evolution and Evaluation*, New York, NY, July 2000. Springer-Verlag.

- [147] Dhrubajyoti Goswami, Ajit Singh, Bruno R. Preiss: From Design Patterns to Parallel Architectural Skeletons. *J. Parallel Distrib. Comput.* 62(4): 669-695 (2002).
- [148] D. Goswami, A. Singh, B. Preiss. "From design patterns to parallel architectural skeletons". *Journal of Parallel and Distributed Computing*, 62(4), pages 669-695, 2002.
- [149] M. M. Akon, D. Goswami and H. F. Li, "SuperPAS: A Parallel Architectural Skeleton Model Supporting Extensibility and Skeleton Composition". Accepted to the Second International Symposium on Parallel and Distributed Processing and Applications (ISPA'04), Hong Kong, 13-15 December, 2004. Appeared in Springer's Lecture notes in Computer Science (LNCS), Vol. 3358, pp. 985-996.
- [150] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. "P3L: A structured high level parallel language and its structured support". *Concurrency: Practice and Experience*, 7(3):225-255, 1995.
- [151] S. Pelagatti, "Task and data parallelism in P3L", In *Patterns and Skeletons For Parallel and Distributed Computing*, F. A. Rabhi and S. Gorlatch, Eds. Springer-Verlag, London, 155-186, 2003.
- [152] J. Serot. "Embodying parallel functional skeletons: an experimental implementation on top of MPI". In C. Lengauer, M. Griebel, and S. Gorlatch, editors, *Proc. of EURO-PAR'97*, Passau, Germany, volume 1300 on LNCS, pages 629-633, Springer-Verlag, Berlin, August 1997.
- [153] Susanna Pelagatti, "Compiling and supporting skeletons on MPP", *Proceedings of MPPM97*, IEEE Computer Society Press, 1997.
- [154] Bruno Bacci, Sergei Gorlatch, Christian Lengauer, Susanna Pelagatti: *Skeletons and Transformations in an Integrated Parallel Programming Environment*. PaCT 1999: 13-27.
- [155] F. Rabhi "Exploiting parallelism in functional languages: A 'paradigm oriented' approach". In *Abstract Machine Models for Highly Parallel Computing*, pages 118-139. Oxford University Press, Oxford, 1995.
- [156] Towards Patterns of Web Services Composition B. Benatallah, M. Dumas, M.C. Fauvet and F.A. Rabhi, Technical Report no UNSW-CSE-0111, School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia, November 2001.
- [157] J. Darlington, Y. Guo, H. W. To, and Y. Jing. "Skeletons for structured parallel composition". In *Proc. of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, volume 30 of SIGPLAN Notices, pages 19-28, July 1995.
- [158] Newhouse, S., Mayer, A., and Darlington, J. 2000. A Software Architecture for HPC Grid Applications (Research Note). In *Proceedings From the 6th international Euro-Par Conference on Parallel Processing* (August 29 - September 01, 2000). A. Bode, T. Ludwig, W. Karl, and R. Wismüller, Eds. *Lecture Notes In Computer Science*, vol. 1900. Springer-Verlag, London, 686-689.

- [159] R. Loogen, Y. Ortega, R. Pena, S. Prebe, F. Rubio. Parallelism Abstractions in Eden. In Patterns and Skeletons For Parallel and Distributed Computing, F. A. Rabhi and S. Gorlatch, Eds. Springer-Verlag, London, 95-128, 2003.
- [160] G. H. Botorog, H. Kuchen. "Skil: An Imperative Language with Algorithmic Skeletons for Efficient Distributed Programming", Proceedings of the Fifth International Symposium on High Performance Distributed Computing (HPDC-5), IEEE Computer Society Press, 243-252, 1996.
- [161] H. Kuchen, M. Cole. "The Integration of Task and Data Parallel Skeletons". Parallel Processing Letters 12(2): 141-155, 2002.
- [162] H. Kuchen, "A Skeleton Library", Technical Report 6/02-I, Angewandte Mathematik und Informatik, University of Münster, 2002.
- [163] H. Kuchen. "A Skeleton Library". Proceedings of Euro-Par 2002, LNCS 2400, 620-629, (C) Springer-Verlag, 2002.
- [164] H. Kuchen, J. Striegnitz. "Higher-Order Functions and Partial Applications for a C++ Skeleton Library". Proceedings of ISCOPE 2002, ACM, 2002.
- [165] The Message Passing Interface (MPI) standard. See website at: <http://www-unix.mcs.anl.gov/mpi/>.
- [166] Murray I. Cole and Andrea Zavanella, "Coordinating Heterogeneous Parallel Systems with Skeletons and Activity Graphs", Journal of Systems Integration, 10(2), pp 127-143, 2001.
- [167] Murray I. Cole, "Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation", Pitman, 1989.
- [168] M. Cole. Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation. MIT Press, 1988.
- [169] Murray Cole. "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming". Parallel Computing, Volume 30, Issue 3, Pages 389-406, March 2004.
- [170] Anne Benoit, Murray Cole, "Two Fundamental Concepts in Skeletal Parallel Programming" International Conference on Computational Science (2) 2005: 764-771.
- [171] A. Benoit, M. Cole, J. Hillston and S. Gilmore, "Flexible Skeletal Programming with eSkel", EuroPar 2005, Lisbon, Portugal, LNCS Series, Springer Verlag. September 2005.
- [172] A. Benoit, M. Cole, S. Gilmore and J. Hillston, "Evaluating the performance of skeleton-based high level parallel programs", In M. Bubak, D. van Albada, P. Sloot and J. Dongarra, editors, The International Conference on Computational Science (ICCS 2004), Part III, LNCS, pages 299-306. Springer Verlag, 2004.

- [173] Marco Vanneschi. "The programming model of ASSIST, an environment for parallel and distributed portable applications". *Parallel Computing*, Volume 28, Issue 12, Pages 1709-1732, December 2002.
- [174] M. Aldinuccia, S. Campab, P. Ciullob, M. Coppolaa, M. Daneluttob, P. Pesciullesib, R. Ravazzolob, M. Torquatib, M. Vanneschiband C. Zoccolob, "ASSIST: A framework for experimenting with structured parallel programming environment design", *ParCo2003: Parallel Computing, Software Technology, Algorithms, Architectures and Applications* (in series: *Advances in Parallel Computing*), G.R. Joubert, W.E. Nagel, F.J. Peters, W.V. Walter, editors. Elsevier, 2004.
- [175] M. Danelutto. "On skeletons and design patterns". In *Parallel Computing, Fundamentals and Applications, Proceedings of the International Conference ParCo99*, Imperial College Press, 2000, pages 460-467.
- [176] Kiminori Matsuzaki, Kazuhiko Kakehi, Hideya Iwasaki, Zhenjiang Hu, Yoshiki Akashi, A Fusion-Embedded Skeleton Library, *International Conference on Parallel and Distributed Computing (EuroPar 2004)*, Pisa, Italy, 31st August - 3rd September, 2004. LNCS 3149, Springer Verlag. pp.644-653 .
- [177] D. Szafron and J. Schaeffer, An Experiment to Measure the Usability of Parallel Programming Systems, *Concurrency Practice and Experience*, Vol. 8, No. 2, March 1996, pp. 147-166.
- [178] G. Geist and V. Sunderam. *Network-Based Concurrent Computing on the PVM System. Concurrency: Practice and Experience*, vol. 4, no. 4, pp. 293-311, 1992.
- [179] David B. Skillicorn, Domenico Talia. "Models and languages for parallel computation". In *ACM Computing Surveys*, Volume 30, Number 2, pages: 123-169, 1998, ACM Press, New York, NY, USA, ISSN:0360-0300.
- [180] Dominique Parquer, "A Survey of Visual Programming Tools", Technical Report, Department of Computing Science, University of Alberta, Canada, July, 2003.
- [181] Skeletons Home Page. See website at <http://homepages.inf.ed.ac.uk/mic/Skeletons/>.
- [182] Borland, "Model Maker Design Patterns - Delphi design pattern", See website at <http://www.modelmakertools.com/modelmaker/design-patterns.html>.
- [183] Lock Design Pattern. A description is available at: <http://www.castle-cadenza.demon.co.uk/lock.htm>.
- [184] Borland's Together Soft. See website at <http://www.borland.com/together/>.
- [185] Berna L. Massingill and Timothy G. Mattson and Beverly A. Sanders, "A Pattern Language for Parallel Application Programs (Research Note)", *Lecture Notes in Computer Science*, volume 1900, 2001.

- [186] Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders; "Additional Patterns for Parallel Application Programs": Proceedings of the Tenth Pattern Languages of Programs Workshop (PloP 2003), 2003.
- [187] Autonomic Computing, IBM research, <http://www.research.ibm.com/autonomic/index.html>.
- [188] Fabio Kon, Fábio Costa, Roy Campbell, and Gordon Blair, "The Case for Reflective Middleware", Communications of the ACM. Vol. 45, No. 6, pp. 33-38. June, 2002.
- [189] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães, and Roy H. Campbell, "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB", IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000). New York. April 3-7, 2000.
- [190] M. Blay-Fornarino, A-M. Pinna-Dery, and M. Riveill, "Towards dynamic configuration of distributed applications", In 2nd IEEE International Workshop on Aspect Oriented Programming for Distributed Computing Systems, ISBN 0-7695-1588-6, Vienna, Austria, pages 487-492, July 2-5 2002.
- [191] Ian Foster, "What Is the Grid? A Three Point Checklist", GRIDtoday, vol. 1, no. 6, 2002. Available at <http://www.gridtoday.com/02/0722/100136.html>
- [192] Nelson Minar, "Distributed Systems Topologies: Part1", Technical Report, December 2001. See website at http://www.openp2p.com/pub/a/p2p/2001/12/14/topologies_one.html.
- [193] Nelson Minar, "Distributed Systems Topologies: Part2", Technical Report, January 2002. See website at http://www.openp2p.com/pub/a/p2p/2002/01/08/p2p_topologies_pt2.html
- [194] The *Structured Query Language (SQL)*. See websites at <http://www.sql.org/> and www.mysql.com/documentation.
- [195] Martin Alt, Jens Muller, and Sergei Gorlatch, "Towards High-Level Grid Programming and Load-Balancing: A Barnes-Hut Case Study", in Proceeding of Euro-Par 2005, J. C. Cunha and P. D. Medeiros (Eds.), September 2005, LNCS 3648, Springer.
- [196] M. Aldinucci, M. Danelutto, J. Dunnweber, and S. Gorlatch, "Optimization Techniques for Implementing Parallel Skeletons In Grid Environments", 4th International Workshop on "Constructive Methods for Parallel Programming" (CMPP 2004), Stirling, Scotland, UK, 14 July 2004.
- [197] Dan Kusnetzky, "DataSynapse: Using Grid Computing to Create a Virtual Environment for Applications", White Paper. Available at: <http://download.intel.com/business/bss/technologies/grid/datasynapse.pdf>.
- [198] "SGI", High performance solutions for business and scientific areas. See website at <http://www.sgi.com>.
- [199] "Gria, Service Oriented Collaborations for Industry and Commerce". See website at <http://www.gria.org/>.

- [200] I. Taylor, M. Shields, I. Wang and R. Philp, "Grid Enabling Applications Using Triana", In Workshop on Grid Applications and Programming Tools. Seattle, 2003.
- [201] Ian Taylor, Ian Wang, Matthew Shields, and Shalil Majithia, "Distributed Computing with Triana on the Grid", *Concurrency and Computation: Practice and Experience*, 17(1-18), 2005.
- [202] Peer-to-Peer simplified. See website at <http://www.p2psimplified.org/>. Last visited: August 2005.
- [203] "Triana User Guide", 2005 – work in progress. Available at: <https://forge.nesc.ac.uk/docman/view.php/33/104/UserGuide.pdf>
- [204] M.Cecilia Gomes, José C. Cunha, and Omer F. Rana, "Pattern-based Configuration and Execution Control of Grid-aware Problem Solving Environments", Poster presentation at the EuroConference on Problem Solving Environments and the Information Society, EU-RESCO Conference on Advanced Environments and Tools for High Performance Computing, Albufeira, Portugal, 2003.
- [205] Kaizar Amin, Gregor von Laszewski, Mihael Hategan, Nestor J. Zaluzec, Shawn Hampton, Albert Rossi, "GridAnt: A Client-Controllable Grid Workflow System", Proceedings of the 37th Hawaii International Conference on System Sciences - HICSS 2004.
- [206] Gregor Laszewski and Mike Hategan, "Grid Workflow - An Integrated Approach", Argonne National Laboratory, 2005. Available from <http://www-unix.mcs.anl.gov/laszewsk/papers>
- [207] Gregor Laszewski and Mike Hategan, "Workflow Concepts of the Java CoG Kit", *Journal of Grid Computing*, 3:3-4, 2005.
- [208] Java Cog Kit, <http://www.globus.org/cog>
- [209] K. Amin and G. von Laszewski and R. Ali and O. Rana and D. Walker, "An Abstraction Model for a Grid Execution Framework", *Euromicro Journal of Systems Architecture*, 2005, accepted for publication.
- [210] Craig A. Lee, B. Scott Michael, "The Use of Content-Based Routing to Support Events, Coordination and Topology-Aware Communication in Wide-Area Grid Environments", in *Process Coordination and Ubiquitous Computing*, Editors Dan C. Marinescu and Craig Lee, CRC Press, pp. 99-118, 2003. ISBN: 0-8493-1470-4.
- [211] George A. Papadopoulos, Farhad Arbab, "Configuration and dynamic reconfiguration of components using the coordination paradigm". *Future Generation Comp. Syst.* 17(8): 1023-1038, 2001.
- [212] Juan Guillen Scholten, Farhad Arbab, Frank S. de Boer, Marcello M. Bonsangue, "A Channel-based Coordination Model for Components". *Electr. Notes Theor. Comput. Sci.* 68(3), 2003.

- [213] Theophilos A. Limniotes, George A. Papadopoulos, Farhad Arbab, "Coordinating Web Services Using Channel Based Communication", COMPSAC 2004: 486-491, 2004.
- [214] George A. Papadopoulos, Farhad Arbab, "Coordination Models and Languages", *Advances in Computers* 46: 330-401, 1998.
- [215] T. W. Malone and K. Crowston, "The interdisciplinary study of coordination", *ACM Computing Surveys*, 26(1):87-119, March 1994.
- [216] Robert Tolksdorf, "Models of Coordination", *Engineering Societies in the Agent World (ESAW 2000)*: 78-92, 200.
- [217] Ozalp Babaoglu, Keith Marzullo, "Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms", in *Distributed Systems*, Editor S. Mullender, Addison-Wesley, pp. 55-96, 1993.
- [218] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill, "Patterns for Parallel Programming", *Software Patterns Series*, Addison Wesley Professional, 2004. ISBN: 0-321-22811-1.
- [219] Paris Avgeriou, "Architectural patterns revisited - a pattern language", *Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPlop 2005)*, Irsee, Germany, July 2005, pp. 1-39.
- [220] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2002.
- [221] "XtremWeb, A Global Computing Experimental Platform". See Web site at: <http://www.lri.fr/fedak/XtremWeb/>.
- [222] "BOINC, Berkeley Open Infrastructure for Network Computing". See Web site at: <http://boinc.berkeley.edu/>.
- [223] D. Abramson, J. Giddy, and L. Kotler, "High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid?", *International Parallel and Distributed Processing Symposium (IPDPS)*, pp 520- 528, Cancun, Mexico, May 2000.
- [224] J. Linderoth, S. Kulkarni, J-P. Goux, and M. Yoder, "An Enabling Framework for Master-Worker Applications on the Computational Grid", *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing (HPDC9)*, August 2000.
- [225] S. Agrawal, J. Dongarra, K. Seymour, and S. Vadhiyar, "NetSolve: Past, Present, and Future - A Look at a Grid Enabled Server", *Making the Global Infrastructure a Reality*, F. Berman, G. Fox, and A. Hey (Eds), Wiley Publishing, 2003.
- [226] Y. Tanaka, H. Nakada, S. Sekiguchi, Suzumura Suzumura, and S. Matsuoka, "Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing", *Journal of Grid Computing*, 1(1):41-51, 2003.

- [227] N. Furmento, W. Lee, A. Mayer, S. Newhouse, and J. Darlington, "ICENI: An Open Grid Service Architecture implemented with Jini", SuperComputing 2002, Baltimore, Maryland, USA, 2002.
- [228] H. Casanova, G. Obertelli, B. Berman, and R. Wolski, "The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid", Supercomputing 2000, IEEE and ACM-SIGARCH, Nov 2000.
- [229] M.Y. Gulamali, A.S. McGough, S. Newhouse, et al , "Using ICENI to run parameter sweep applications across multiple Grid resources", Global grid forum 10, Case studies on grid applications workshop, Berlin, Germany, March 2004, 2004.
- [230] S. Androutsellis-Theotokis, and D. Spinellis, "A Survey of Peer-to-Peer Content Distribution Technologies", ACM Computing Surveys (CSUR), Volume 36 , Issue 4, pp. 335-371, December 2004.
- [231] "Gnutella", See Web site at: <http://www.gnutella.wego.com/>.
- [232] I. Clarke, O. Sandberg, B. Wiley, and T.W. Hong, "Freenet: A Distributed Anonymous Information Storage and Retrieval System", Designing Privacy Enhancing Technologies, Lecture Notes in Computer Science 2009, H. Federrath (Ed.), Springer-Verlag, Berlin, 2001, pp. 46-66.
- [233] "Entropia", See Web site at: <http://www.entropia.com/>
- [234] A. Iamnitchi, I. Foster, and D. Nurmi, "A Peer-to-Peer Approach to Resource Location in Grid Environments", Symp. on High Performance Distributed Computing, Aug. 2002.
- [235] I. Foster, and Adriana Iamnitchi, "On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing", IPTPS 2003, pp. 118-128.
- [236] M. Grand, "Patterns in Java , Volume 1", John Wiley and Sons, 1998.
- [237] E. C. Wyatt, P. O'Leary, "Interactive Poster: Grid-Enabled Collaborative Scientific Visualization Environment", 15th IEEE Visualization 2004 Conference (VIS 2004), IEEE Computer Society, 2004.
- [238] F. Geysermans, and J. Miller, "Build asynchronous applications with the Distributed Event-Based Architecture for Web Services", Distributed Event-Based Architecture (DEBA) from IBM Corporation. Available at: <http://www-128.ibm.com/developerworks/webservices/library/ws-dbarch/?Open&ca=daw-ws>.
- [239] "Publish-Subscribe Notification for Web services", contributors: IBM, Akamai Technologies, Computer Associates, Fujitsu Laboratories of Europe, Globus, Hewlett-Packard, SAP AG, Sonic Software, TIBCO Software. Whitepaper available at: <http://www-128.ibm.com/developerworks/library/specification/ws-pubsub/>.

- [240] B. Tierney, B. Crowley, D. Gunter, M. Holding, J. Lee, and M. Thompson, "A Monitoring Sensor Management System for Grid Environments", *Journal of Cluster Computing* 4, 1, Kluwer Academic Publishers, 19-28, Mar. 2001.
- [241] M. Baker, and M. Grove "jGMA: A lightweight implementation of the Grid Monitoring Architecture", proceedings of the UK e-Science All Hands Conference 2004. See Web site at: <http://www.allhands.org.uk/2004/>.
- [242] A. Waheed, W. Smith, J. George, and J.C. Yan, "An Infrastructure for Monitoring and Management in Computational Grids", *Proceedings of LCR '00: Selected Papers from the 5th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, Springer-Verlag, 2000.
- [243] A. Fuggetta, G.P. Picco, and G. Vigna, "Understanding code mobility", *IEEE Transactions on Software Engineering*, Volume 24(5), pp. 342-361, 1998.
- [244] Java Applets, Sun Microsystems. See website at <http://java.sun.com/applets/>.
- [245] P.D. Coddington, L. Lu, D. Webb, and A.L. Wendelborn, "Extensible Job Managers for Grid Computing", *Twenty-Sixth Australasian Computer Science Conference (ACSC2003)*, Michael J. Oudshoorn (Ed.), *Conferences in Research and Practice in Information Technology* 2003, Australian Computer Society, pp. 151-159, 2003.
- [246] O. F. Rana, and Luc Moreau, "Issues in Building Agent-Based Computational Grids", *UK Multi-Agent Systems Workshop*, Oxford, December 2000.
- [247] A. Jhoney, M. Kuchhal, and Venkatakrishnan, "Grid Application Framework for Java (GAF4J)", Technical report, IBM Software Labs, India, 2003. Available at <https://secure.alphaworks.ibm.com/tech/GAF4J>
- [248] , K. Amin, G. von Laszewski, R. Ali, O. Rana, and D. Walker, "An Abstraction Model for a Grid Execution Framework", *Euromicro Journal of Systems Architecture*, 2005.
- [249] G. Aloisio, M. Cafaro, P. Falabella, C. Kesselman, R. Williams, "Grid Computing on the Web Using the Globus Toolkit", *HPCN Europe* 2000, pp. 32-40, 2000.
- [250] G. von Laszewski, J. Gawor, P. Lane, N. Rehn, M. Russell, and K. Jackson, "Features of the Java Commodity Grid Kit", *journal of Concurrency and Computation: Practice and Experience*, 14:1045, 2002.
- [251] M. Sato, K. Kusano, H. Nakada, S. Sekiguchi, and S. Matsuoka, "netCFD: a Ninf CFD component for Global Computing, and its Java applet GUIH", *proceedings of HPC Asia* 2000, pp. 501-506.
- [252] S. W. Loke, "Towards Data-Parallel Skeletons for Grid Computing: An Itinerant Mobile Agent Approach", *proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID)*, 2003.

- [253] "MonALISA: An Agent based, Dynamic Service System to Monitor, Control and Optimize Grid based Applications", I.C.Legrand, H.B.Newman, California Institute of Technology, Pasadena, CA 91125, USA R.Voicu, European Center for Nuclear Research . CERN, Geneva, Switzerland C.Cirstoiu, C.Grigoras, M.Toarta, C. Dobre, Polytechnic University Bucharest, Romania CHEP 2004, Interlaken, Switzerland, September 2004. Available at: <http://monalisa.caltech.edu>.
- [254] H. Casanova, and J. Dongarra, "NetSolve: A network server for solving computational science problems", proceedings of Supercomputing 96.
- [255] H. Casanova, and J. Dongarra, "Using agent-based software for scientific computing in the NetSolve system", Parallel Computing, 24(1998).
- [256] B. Meyer, "Applying «Design by Contract»", in journal Computer, IEEE, 25, 10, October 1992, pp. 40-51. Available at <http://se.ethz.ch/meyer/publications/computer/contract.pdf>.
- [257] Douglas C. Schmidt, Frank Buschmann "Patterns, Frameworks, and Middleware: Their Synergistic Relationships", In Proceedings of the 25th international Conference on Software Engineering (Portland, Oregon, May 03 - 10, 2003). International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 694-704.
- [258] Weiguo Liu and Bertil Schmidt, "A Case Study on Pattern-based Systems for High Performance Computational Biology", IPDPS 2005.
- [259] A. Benoit, M. Cole, S. Gilmore, and J. Hilston, "Scheduling Skeleton-Based Grid Applications Using PEPA and NWS", Comput. J. 48(3): 369-378 (2005).
- [260] M. Aldinucci, M. Danelutto, J. D nnweber, and S. Gorlatch. "Optimization techniques for skeletons on grid". In L. Grandinetti, editor, Grid Computing and New Frontiers of High Performance Processing, Advances in Parallel Computing. Elsevier, 2005. Available as CoreGRID TR-0001.
- [261] Seng Wai Loke, "Towards Data-Parallel Skeletons for Grid Computing: An Itinerant Mobile Agent Approach". CCGRID 2003, CCGRID 2003: 651-.
- [262] Sun GridEngine. Web site: <http://gridengine.sunsource.net>.
- [263] Announcement of the DRMAA C Binding for Sun GridEngine: <http://gridengine.sunsource.net/news/DRMAA0dot8-announce.html>.
- [264] J. Herrera, E. Huedo, R. Montero, I. Llorente, "Developing Grid-Aware Applications with DRMAA on Globus-Based Grids". Euro-Par 2004, pp:429-435.
- [265] B. Nitzberg, J. M. Schopf, and J. Jones, "PBS Pro: GRID Computing and Scheduling Attributes", In "Grid resource management: state of the art and future trends", Kluwer Academic Publishers, 2004, ISBN:1-4020-7575-8 .

- [266] Ian Wang, "P2PS (Peer-to-Peer Simplified)", Proceedings of 13th Annual Mardi Gras Conference - Frontiers of Grid Applications and Technologies. Louisiana State University, pages 54-59, February 2005.
- [267] S. Thatte, T. Andrews, F. Curbera, H. Dholakia, Y. Golland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, I. Trickovic, and S. Weerawarana, Business Process Execution Language for Web Services Version 1.1 , 2003. Available at: <http://www.ibm.com/developerworks/library/specification/ws-bpel/>
- [268] The Condor Project. See website at <http://www.cs.wisc.edu/condor/>.
- [269] I. Taylor, M. Shields, and I. Wang, "Resource Management for the Triana Peer-to-Peer Services", In Jarek Nabrzyski, Jennifer M. Schopf, and Jan Weglarz, editors, Grid Resource Management, pages 451-462. Kluwer Academic Publishers, 2004.
- [270] "Extended Backus-Naur Form (EBNF)", International Organization for Standardization, <http://dret.net/biblio/reference/iso14977>.